Virtualna memorija

1 Uvod

U prethodnom poglavlju smo predstavili o različite strategijame upravljanja memorijom koje se koriste u računarskim sistemima. Sve ove strategije imaju isti cilj: održavati višestruke procese u memoriji istovremeno kako bi se omogućilo multi-programiranje. Međutim, oni obično zahtevaju da čitav proces bude u memoriji pre nego što se može izvršiti.

Virtualna memorija je tehnika koja omogućava izvršavanje procesa koji nisu u potpunosti u memoriji. Glavna prednost ove šeme je da programi mogu biti čak i veći od raspoložive fizičke memorije. Nadalje, virtualna memorija abstrakuju glavnu memoriju u ekstremno veliki, kontinualni niz lokacija, odvajajući pri tome logičku memoriju, onako kako je korisnik vidi, od fizičke memorije. Ova tehnika oslobađa programere od brige u vezi sa ograničenošću memorije. Virtualna memorija takođe omogućava procesima da lako dele datoteke i implementiraju deljenu memoriju. Dodatno, omogućava efikasan mehanizam za kreiranje procesa. Međutim, virtualnu memoriju nije lako implementirati i performanse sistema se mogu degradirati višestruko ukoliko se to uradi pogrešno.

Algoritmi za upravljanje memorijom navedeni u ranije potrebni su zbog jednog osnovnog zahteva: instrukcije koje se izvršavaju moraju biti smeštene u fizičkoj memoriji. Prvi način da se ovaj zahtev zadovolji jeste da se celokupni logički prostor smešta u fizičku memoriju. Dinamičko učitavanje može pomoći da se ublaži ovo ograničenje, ali, generalno, to obično zahteva posebne mere predostrožnosti i dodatni rad programera.

Uslov da instrukcije moraju biti u fizičkoj memoriji jeste obavezan i razuman, ali on takođe ograničava veličinu programa koji se izvršava na veličinu fizičke memorije. Sa druge strane, istraživanja sprovedena na u realnom programima pokazuju da u mnogim slučajevima celokupan program nije neophodan, u ovom smislu. Na primer, uzmite u obzir sledeće primere:

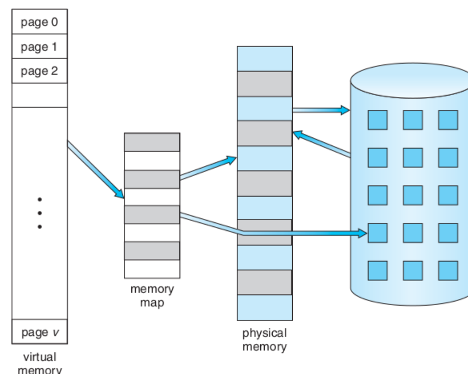
- Programi često imaju deo kod za manipulisanje neobičnim greškama koje se mogu javiti u sistemu. Pošto se ove greške retko, ako ikada, pojave u praksi, ovaj se kod gotovo nikada ne izvršava;

- Nizovima, listama i tabelama često je dodeljeno više memorije nego što im je zapravo potrebno: niz se može deklarisati tako da sadrži od 10000 elemenata, iako se retko zaista koristi više od od 100 elemenata;

- Određene opcije i funkcije programa mogu se izuzetno retko koristiti.

Čak i u onim slučajevima kada je potreban ceo program, možda svi delovi programa neće biti potrebni istovremeno. Mogućnost izvršavanja programa koji je samo delimično u memoriji pružila bi mnoge prednosti:

- Program više neće biti ograničen količinom dostupne fizičke memorije. Korisnici bi mogli pisati programe smatrajući da im je dostupan izuzetno veliki (virtualni) adresni prostor, što znatno pojednostavljuje zadatak programiranja;



Slika 1: Virtualna memorija omogućava izvršavanje programa većih od dostupne fizičke memorije

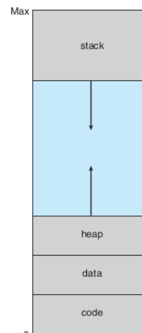
- Budući da bi svaki korisnički program mogao da zauzme manje fizičke memorije, više programa se može istovremeno pokrenuti, što će rezultovati odgovarajućim povećanjem iskorišćenosti procesora kao i propusnost (broj procesa koji se kompletiraju u jedinici vremena), ali bez povećanja vremena odziva ili ukupnog vremena izvršavanja;
- Bilo bi potrebno manje U/I operacija za učitavanje ili zamenu korisničkih programa u memoriju, tako da bi svaki korisnički program radio brže.

Na taj način, pokretanje programa koji nije u potpunosti u memoriji koristilo bi kako sistemu, tako i korisniku. Kao što smo rekli, koncept virtualne memorija podrazumeva razdvajanje logičke memorije (onakve kakvu je korisnik vidi) od fizičke memorije. Ovo razdvajanje omogućava programerima ekstremno veliku virtualnu memoriju, čak i u slučajevima kada je na raspolaganju znatno manja fizička memorija (slika 1).

Virtualna memorija znatno olakšava zadatak programiranja, jer programer više ne mora da brine o količini dostupne fizičke memorije - umesto toga može da se koncentriše na problem koji će rešavati.

Virtualni adresni prostor procesa odnosi se na logički (ili virtualni) prikaz načina na koji se proces čuva u memoriji. Tipično ovo podrazumeva da proces počinje na određenoj logičkoj adresi, recimo adresi 0, i zauzima kontinualne memorijske lokacije, kao što je prikazano na slici 2. Međutim, kao što smo diskutovali ranije, fizička memorija se u stvarnosti može organizovati u okvire na način da fizički okviri dodeljeni nekom procesu uopšte nisu povezani (kontinualni). Uloga *jedinice za upravljanje memorijom* (MMU) jeste da mapira logičke stranice u fizičke okvire u memoriji.

Primetite na slici 2 da dozvoljavamo *heap* memoriji da se širi jer se ona koristi za dinamičku dodelu memorije. Slično tome, dopuštamo *steku* da se povećava u memoriji, na primer putem sukcesivnih poziva funkcija. Veliki prazan prostor (ili šupljina) između heap memorije i steka je deo virtualnog adresnog prostora,

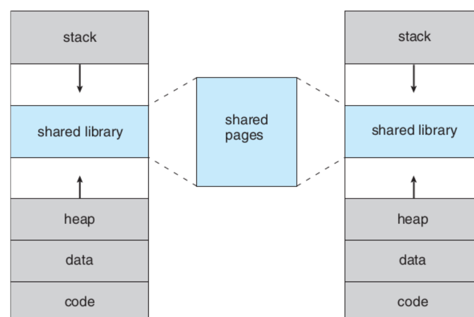


Slika 2: Virtualni adresni prostor

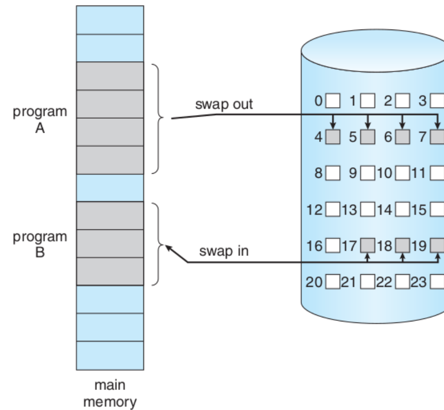
ali će trebati stvarne fizičke okvire samo ukoliko se heap ili stek budu širili. Virtualni adresni prostori koji podrazumevano uključuju u sebe šupljine poznati su kao *proređeni adresni prostori* (eng. *sparse address spaces*). Korištenje proređenog adresnog prostora je korisno jer se šupljine mogu smanjivati kako segmenti heap memorije ili steka rastu ili u slučajevima kada želimo dinamički povezivati biblioteke (ili možda druge deljene objekte) tokom izvršavanja programa.

Dakle, pored razdvajanja logičke memorije od fizičke memorije, virtualna memorija omogućava deljenje datoteka i memorije između dva ili više procesa putem deljenja stranica. To dovodi do sledećih prednosti:

- Sistemske biblioteke mogu se deliti od strane nekoliko procesa mapiranjem deljenog objekta u virtualni adresni prostor. Iako svaki proces smatra da su biblioteke deo njegovog virtualnog adresnog prostora, stvarni okviri u kojima su biblioteke smeštene u fizičkoj memoriji dele između svih procesa (slika 3). Tipično se biblioteka mapira tako da je moguće samo njeno čitanje (*read-only mode*) u adresnom prostoru svakog procesa koji je sa njom povezan



Slika 3: Deljene biblioteke korišćenjem virtualne memorije



Slika 4: Transfer stranica memorije u sekundarnu memoriju i nazad

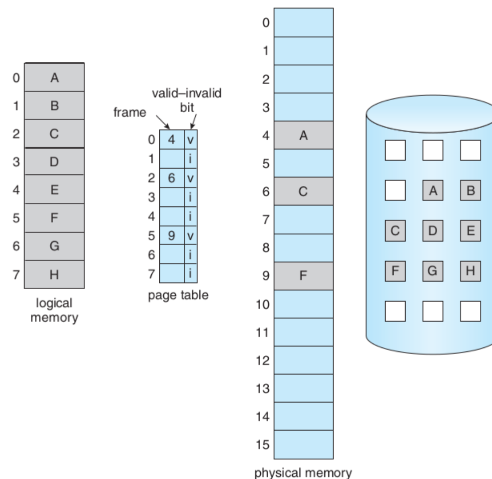
- Slično tome, procesi mogu da dele memoriju. Već smo pričali ranije da dva ili više procesa mogu komunicirati korišćenjem deljene memorije. Virtualna memorija omogućava jednom procesu da stvori region memorije koji može da deli sa drugim procesom. Procesi koji dele ovaj region smatraju ga delom svog virtualnog adresnog prostora, ali se stvarni fizički okviri memorije dele, slično kao što je prikazano na slici 3

- Stranice se mogu deliti tokom kreiranja procesa sistemskim pozivom $fork()$, čime se ubrzava kreiranje procesa.

U nastavku detaljnije istražujemo ove i druge prednosti virtuelne memorije. Najpre ćemo, međutim, predstaviti primenu virtualne memorije kroz *stranice na zahtev* (eng. *demand paging*).

2 Stranice na zahtev (Demand Paging)

Razmislite o tome na koji način se izvršni program može učitati sa diska u memoriju. Jedna od opcija je učitavanje celog programa u fizičku memoriju u vreme izvršavanja programa. Međutim, problem sa ovim pristupom je što nam u početku možda neće biti potreban čitav program u memoriji. Pretpostavimo da program počinje listom dostupnih opcija od kojih će korisnik izabrati jednu. Učitavanje celog programa u memoriju rezultuje učitavanjem izvršnog koda za *sve* opcije, bez obzira da li je data opcija na kraju uopšte odabrana od strane korisnika. Alternativna strategija je učitavanje stranica samo onda kada su potrebne. Ova tehnika je poznata kao *stranice na zahtev* i obično se koristi u sistemima virtualne memorije. Sa virtualnom memorijom koja koristi stranice na zahtev, stranice se učitavaju samo kada se zahtevaju tokom izvršavanja programa. Stranice kojima se nikada ne pristupa nikada se neće ni učitavati u fizičku memoriju.



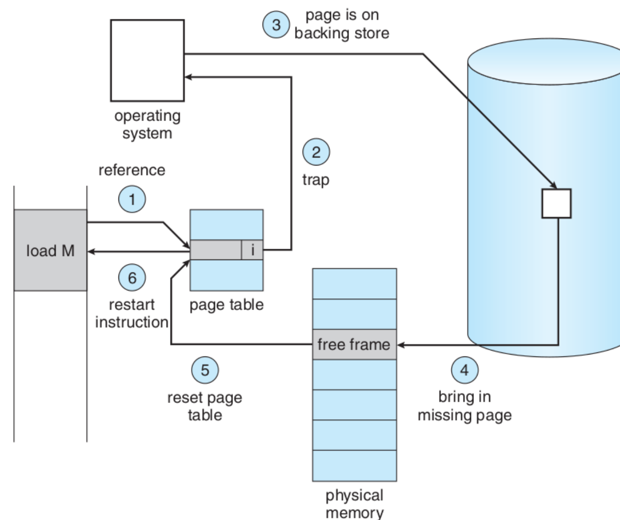
Slika 5: Tabela stranica kada nisu sve strance u osnovnoj memoriji

Sistem sa stranicama na zahtev sličan je sistemu straničenja sa zamenjivanjem (slika 4), gde se procesi nalaze u sekundarnoj memoriji (obično na disku). Kada želimo da izvršimo neki proces, zamenimo ga u memoriju. Međutim, umesto da izvršimo zamenu čitavog procesa u memoriju, sada koristimo *delimičnu zamenu* (eng. *lazy swap*). Delimična zamena nikada ne prebacuje stranicu u memoriju osim ako će ta stranica biti potrebna.

2.1 Osnovni koncept

Kada treba učitati proces u memoriju, sistem za zamenu pretpostavlja koje će se stranice procesa koristiti pre nego se proces ponovo zameni nazad u sekundarnu memoriju. Umesto da učita čitav proces, sistem za zamenu učitava samo te stranice u memoriju. Na taj način se izbegava učitavanje memorijskim stranicama koje se ionako neće koristiti, smanjuje se vreme zamene i količina potrebne fizičke memorije.

Uz pomoć ove šeme, potreban nam je neki oblik hardverske podrške da bismo razlikovali stranice koje se nalaze u memoriji i stranice koje se nalaze na disku. Bit validnosti opisan u prethodnoj lekciji može se koristiti u tu svrhu. Ovog puta, međutim, kada je ovaj bit postavljen na „validno", pridružena stranica je i validna i rezidentna u memoriji. Ako je bit postavljen na "ne-validno", stranica ili nije validna (tj. nije u logičkom adresnom prostoru procesa) ili je validna, ali se trenutno nalazi na disku. Unos u tabeli stranica za stranicu koja se učitana u memoriju je postavljen kao i obično, ali unos u tabeli stranica za stranicu koja trenutno nije u memoriji ili je jednostavno postavljen na nevalidni ili sadrži adresu te stranice na disku. Da bi se ovakav pristup koristio, potrebno je održavati posebnu internu tabelu u kojoj se nalaze sve stranice koje



Slika 6: Obrada izuzetka greške stranice

su dodeljene datom procesu, kako bi se u slučaju da je bit validnosti postavljen na vrednost ne-validna, moglo utvrditi koji je razlog za to. Ova situacija je prikazana na slici 5.

Treba primetiti da označavanje stranice kao ne-validne neće imati nikakvog efekta u slučaju da proces nikada ne pokuša pristupiti toj stranici. Dakle, ako inicijalna pretpostavka bude ispravna i učitamo samo stranice koje su zapravo potrebne, proces će se izvršavati na isti način kako bi se izvršavao i da smo učitali sve stranice. Sve dok se proces izvršava i pristupa stranicama koje su *rezidentne u memorije*, izvršavanje se odvija normalno.

Ali šta se dešava ako proces pokuša da pristupi stranici koja nije učitana u memoriju? Pristup stranici označenoj kao ne-validna uzrokuje grešku stranice (eng. *page fault*). Hardver za straničenje, prilikom transliranja adrese korišćenjem tabele stranica, primetiće da je postavljen bit validnosti postavljen na vrednost ne-validan, što će dovesti do izuzetka u operativnom sistemu. Ovaj izuzetak je posledica toga što željena stranica nije učitana u memoriju. Rutina za obradu ovog izuzetka greške stranice je jednostavna (slika 6):

1. Proveravamo internu tabelu (koja se obično čuva sa kontrolnim blokom procesa) da bismo utvrdili da li referencirana memorija uopšte pripada tom procesu ili je u pitanju ilegalan pristup memoriji drugog procesa

2. Ako je referenca ilegalna, prekinemo proces. Ako nije, ali još tražena stranica nije učitana u memoriju, učitaj je

3. Pronalazimo slobodan okvir u osnovnoj memoriji (na primer, uzimanjem jednog iz liste slobodnih okvira)

4. Zakazujemo operaciju pristupa disku kako bismo učitali željenu stranicu u novo-dodeljeni okvir

5. Kada je čitanje sa diska završeno, ažuriramo internu tabelu i tabelu stranica kako bi označili da je stranica sada u memoriji

6. Ponovo pokrećemo instrukciju koja je i generisala izuzetak. Proces sada može da pristupi stranici kao da je ona sve vreme bila u memoriji.

U ekstremnom slučaju, možemo da započnemo izvršavanje procesa bez ijedne stranice u memoriji. Kada operativni sistem postavi programski brojač na prvu instrukciju procesa, koja se nalazi u okviru nerezidentne memorijske stranice (okriva), to odmah rezultuje greškom stranice. Nakon što se ova stranica učita u memoriju, proces se nastavlja izvršavati, po potrebi generišući nove greške stranica sve dok svaka stranica koja mu je zaista potrebna ne bude rezidentna u memoriji. Od tog trenutka, proces se može izvršavati bez narednih grešaka stranica. Ova šema se naziva *stranice isključivo na zahtev* (eng. *pure demand paging*) - nikad ne učitavaj stranicu u memoriju dok ona nije neophodna.

Hardver za podršku stranicama na zahtev je isti kao i hardver za straničenje i zamenu:

- Tabela stranica - Ova tabela ima mogućnost da označi unos nelegalnim putem bita validnosti ili eventualno specijalnih vrednosti zaštitnih bita;

- Sekundarna memorija - sadrži one stranice koje nisu u glavnoj memoriji. Sekundarna memorija je obično disk velike brzine. Poznat je kao uređaj zamene (eng. *swap device*), a deo diska koji se koristi u tu svrhu poznat je kao *prostor za zamenu* (eng. *swap space*).

Ključni uslov kod implementacije stranica na zahtev jeste mogućnost da se ponovo izvrši bilo koja instrukcija nakon izuzetka greške stranice. Budući da sačuvamo stanje (registre, programski brojač,..) prekinutih procesa kada se dogodi greška stranice, moramo biti u mogućnosti da ponovo pokrenemo proces na potpuno istom mestu i pri identičnom stanju, osim što je sada željena stranica u memoriji i dostupna je. U većini slučajeva ovaj zahtev je lako ispuniti. Greška stranice može se pojaviti prilikom pristupa bilo kojoj memorijskoj lokaciji. Ako se greška stranice dogodi pri prihvatu instrukcije, možemo je ponovo izvršiti ponovnim prihvatom instrukcije. Ako se greška stranice dogodi prilikom prihvata operanda, moramo ponovo prihvatiti i dekodirati instrukciju, a zatim preuzeti operand.

2.2 Performanse stranica na zahtev

Korišćenje stranica na zahtev može značajno uticati na performanse računarskog sistema. Da bismo videli zašto, izračunajmo efektivno vreme pristupa memoriji u slučaju sistema koji koristi stranice na zahtev. Za većinu računarskih sistema vreme pristupa memoriji, označeno ma, kreće se od 10 do 200 nanosekundi. Sve dok nemamo greške stranice, efektivno vreme pristupa je

jednako vremenu pristupa memoriji. Ako se, međutim, dogodi greška stranice, prvo moramo pročitati odgovarajuću stranicu s diska, a zatim pristupiti željenoj memorijskoj reči.

Neka je p verovatnoća greške stranice ($0 \leq$ p ≤ 1). Za očekivano je da će p biti blisko nuli, tj. da imamo znatno manje grešaka stranice u odnosu na situacije kada su stranice već dostupne prilikom referenciranja memorije. Efektivno vreme pristupa je tada

$$\text{efektivno vreme pristupa} = (1 \text{ - } p) \times ma + p \times \text{vreme greške stranice}$$

Da bismo izračunali efektivno vreme pristupa, moramo znati koliko je vremena potrebno da se reši greška stranice. Greška stranice uzrokuje sledeći niz koraka:

1. Izuzetak operativnog sistema;
2. Sačuvati korisničke registre i stanje procesa;
3. Utvrditi da je uzrok izuzetka bio greška stranice;
4. Proveriti da li je referenca stranice legalna i utvrditi lokaciju stranice na disku;
5. Zahtevati učitavanje sa diska u slobodni okvir
   (a) Sačekati u redu čekanja za ovaj uređaj dok se zahtev za čitanje ne servisira
   (b) Sačekati vreme potrebno da uređaj pronađe željene podatke
   (c) Započeti prenos stranice u slobodni okvir
6. Dok se čeka, dodeliti CPU nekom drugom korisniku (raspoređivanje CPU-a, opciono);
7. Primiti prekid od strane U/I podsistema diska (U/I operacija je završena);
8. Sačuvati registre i stanje procesa drugog korisnika (ako je izvršen korak 6);
9. Utvrditi da je izvor prekida prekid U/I podsistem diska;
10. Ažurirati tabelu stranica i internu tabelu kako bi se jasno objavilo da je željena stranica sada u memoriji;
11. Sačekati da se procesor ponovo dodeli ovom procesu;
12. Vratiti korisničke registre, stanje procesa i novu tabelu stranica, a zatim nastaviti sa izvršavanjem prekinute instrukcije.

Nisu svi ovi koraci neophodni u svakom slučaju. Na primer, pretpostavljamo da je u koraku 6 CPU dodeljen drugom procesu dok se izvršava U/I operacija. Ovaj aranžman omogućava multi-programiranje kako bi se održala iskorišćenost CPU-a, ali zahteva dodatno vreme da se nastavi rutina servisiranja grešake stranice kada U/I transfer bude završen.

U svakom slučaju, suočeni smo sa tri glavne komponente vremena servisiranja greške stranice:

1. Servisirati izuzetak greške stranice;
2. Učitati stranicu;
3. Ponovo pokrenuti proces.

Prvi i treći zadatak mogu se, pažljivim kodiranjem, svesti na nekoliko stotina instrukcija. Svaki od ova dva zadatka može trajati od 1 do 100 mikrosekundi. Međutim, vreme učitavanja stranice verovatno će trajati blizu 8 milisekundi. Tipični hard-disk ima prosečno kašnjenje rotacije (obrtanje glave diska) od 3 milisekunde, vreme traženja podataka 5 milisekundi (pomeranje ruke diska do željenog cilindra) i vreme prenosa 0,05 milisekundi. Dakle, ukupno vreme učitavanja stranice je oko 8 milisekundi, uključujući vreme hardvera i softvera. Pri tome, obratite pažnju da smo gledali samo vreme rada uređaja. Ako proces čeka u redu čekanja za dati uređaj, moramo uračunati i vreme čekanja na uređaj tj. vreme dok čekamo da sekundarna memorija bude slobodna da servisira naš zahtev, što dodatno povećava vreme zamene.

Za prosečno vreme servisiranja greške stranice 8 milisekundi i vreme pristupa memoriji od 200 nanosekundi, efektivno vreme pristupa u nanosekundama je

$$\text{efektivno vreme pristupa} = (1 - p) \times (200\text{ns}) + p \text{ x } (8 \text{ ms}) = (1 - p) \times 200 + p \times 8.000.000 = 200 + 7{,}999{,}800 \times p$$

Vidimo da je efektivno vreme pristupa direktno proporcionalno verovatnoći greške stranice. Ako jedan pristup od 1.000 izazove grešku stranice, efektivno vreme pristupa je 8,2 mikrosekunde. Računar će, u tom slučaju, biti usporen faktorom 40 zbog stranica na zahtev! Ako želimo da degradacija performansi bude manja od 10 procenata, verovatnoću greške stranice moramo držati na nivou:

$220 > 200 + 7{,}999{,}800 \times p$,
$220 > 7{,}999{,}800 \times p$,
$p < 0{,}0000025$

To jest, da bi se degradacija performanski u smislu brzine usled straničenja održala na razumnom nivou, moramo dozvoliti manje od jednog pristupa memoriji sa greškom na stranici na 399 990 pristupa memoriji. Sve u svemu, očigledno je važno zadržati nisku stopu grešaka stranica u sistemu sa stranicama na zahtev. U suprotnom, efektivno vreme pristupa povećava se rapidno, dramatično usporavajući izvršavanje procesa.

Dodatni aspekt stranica na zahtev je manipulisanje i, uopšte, upotreba *prostora za zamenu.* Disk U/I namenjen kao prostor za zamenu obično je znatno

brži od onog u kojem se drži fajl-sistem. Razlog za to je činjenica da je u ovom prostoru memorija dodeljena u mnogo većim blokovima, a pretraživanje datoteka i metode indirektne alokacije (pričaćemo o njima na narednom predavanju) se ne koriste. Sistem može, kao rezultat, obezbediti bolji propusni opseg prilikom rada sa stranicama, tako što kopira celokupnu izvršnu datoteku u prostor za zamenu pri pokretanju procesa, a zatim uslužuje zahteve za stanicama korišćenjem prostora za zamenu. Druga opcija je da se zahtevaju stranice iz fajl sistema na početku, ali da se stranice upisuju u prostor za zamenu (*swap space*) kada bude došlo do zamene. Ovaj pristup će omogućiti da se iz fajl sistema čitaju samo potrebne stranice, ali da se sva naredna upisivanja stranica izvršavaju iz prostora za zamenu.

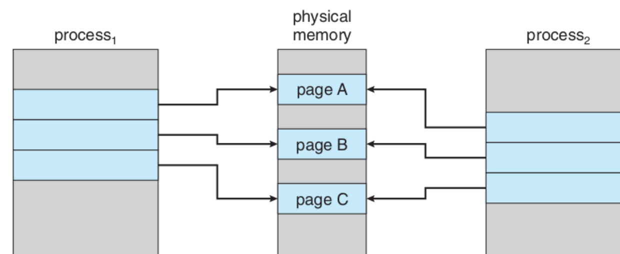
Neki sistemi pokušavaju ograničiti količinu korišćenog prostora za zamenu korišćenjem specifične implementacije stranica na zahtev u slučaju binarnih (izvršnih) datoteka. Stranice za takve datoteke učitavaju se direktno iz fajl sistema. Međutim, kada se zahteva zamena stranice, odgovarajući okviri se mogu jednostavno prebrisati (jer se nikada ne menjaju obzirom da je u pitanju izvršni kod), a stranice se po potrebi ponovo mogu učitati iz fajl sistema. Koristeći ovaj pristup, fajl sistem sam po sebi služi kao skladišteni prostor. Međutim, prostor za zamenu (*swap*) mora se i dalje koristiti u slučaju stranica koje nisu povezane sa saomo datotekom (memorija poznata kao *anonimna memorija*). Ove stranice uključuju stek i heap memoriju za dati proces. Čini se da je ova metoda dobar kompromis i koristi se na nekolicini sistema, uključujući Solaris i BSD Unix.

Mobilni operativni sistemi obično ne podržavaju zamenu. Umesto toga, ovi sistemi zahtevaju stranicu od fajl sistema i oduzimaju aplikacijama stranice koje se mogu samo čitati (kao što je kod), ako količina dostupne memorija bude kritična. Takvi stranice mogu se opet zahtevati od strane fajl sistema ako budu potrebne kasnije. U slučaju iOS-a stranice anonimne memorije nikada se ne oduzimaju od aplikacije, osim ako se aplikacija terminira ili sama eksplicitno ne oslobodi memoriju.

3 Kopiranje pri upisu (Copy-on-write)

Ranije smo ilustrovali kako proces može brzo da započne izvršavanje stranicom na zahtev za stranicu koja sadrži prvu instrukciju. Međutim, kreiranje procesa pomoću sistemskog poziva *fork*() može inicijalno izbeći potrebu za stranicama na zahtev koristeći tehniku sličnu deljenju stranica. Ova tehnika omogućava brzo kreiranje procesa i minimizira broj novih stranica koje moraju biti dodeljene novostvorenom procesu.

Podsetimo se da sistemski poziv *fork*() stvara dete proces koji je duplikat svog roditelja. Tradicionalno, *fork*() je radio tako što je kreirao kopiju adresnog prostora roditelja za dete, duplirajući stranice koje pripadaju roditelju. Međutim, obzirom na to da mnogi dete procesi pozivaju sistemski poziv *exec*() odmah po kreiranju, kopiranje adresnog prostora roditelja može biti nepotrebno. Umesto toga, možemo koristiti tehniku poznatu kao *kopiranje pri upisu*, koja funkcioniše tako što dozvoljava roditelj procesima i dete procesima da u početku

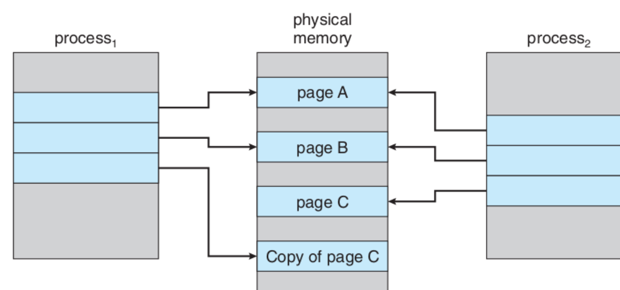


Slika 7: Sadržaj memorije pre nego što proces 1 modifikuje stranicu C

dele iste stranice. Ove zajedničke stranice označene su kao stranice za *kopiranje pri upisu* (*copy-on-write*), što znači da ako bilo koji proces izvrši upis u tu stranicu, stvoriće se kopija te zajedničke stranice. Kopiranje pri upisu ilustrovano je na slikama 7 i 8, koje prikazuju sadržaj fizičke memorije pre i nakon što proces 1 modifikuje stranicu C.

Na primer, pretpostavimo da dete proces pokuša da modifikuje stranicu koja sadrži delove steka, pri čemu je stranica označena za kopirati pri upisu. Operativni sistem će kreirati kopiju ove stranice, preslikavajući je u adresni prostor dete procesa. Tada će dete proces modifikovati kopiranu stranicu, a ne stranicu koja pripada roditelj procesu. Očito, kada se koristi tehnika kopiranja pri upisu, kopiraju se samo stranice koje su modifikovane od strane bilo kojeg procesa - sve neizmenjene stranice mogu se i dalje deliti između roditeljskog i dete procesa. Takođe, očigledno je da samo stranice koje se mogu modifikovati moraju biti tada označene za kopiranje pri upisu. Stranice koje ne mogu da se menjaju (npr stranice koje sadrže izvršni kod) mogu bezbedno da se dele između roditelja i deteta. Kopiranje pri upisu je uobičajena tehnika koju koristi nekoliko operativnih sistema, uključujući Windovs XP, Linux i Solaris.

Kada se utvrdi da će se stranica kopirati koristeći kopiranje pri upisu, važno je zabeležiti lokaciju sa koje će slobodne stranice biti dodeljene. Mnogi opera-



Slika 8: Sadržaj memorije nakon što proces 1 modifikuje stranicu C

tivni sistemi pružaju *domen* (eng. *pool*) slobodnih stranica za takve zahteve. Ove slobodne stranice se obično dodeljuju kada se stek ili heap datog procesa mora proširivati ili kada postoje stranice označene za kopiranje pri upisu. Operativni sistemi obično alociraju ove stranice uz *popunjavanje nulama pri zahtevu*, što znači da su u te stranice upisane nule pre nego što su stranice dodeljene, čime se briše njihov prethodni sadržaj.

Nekoliko verzija Unix OS (uključujući Solaris i Linux) obezbeđuju varijaciju sistemskog poziva *fork*() poznatu kao *vfork*() (fork za virtualne memorije). Ovaj sistemski poziv deluje drugačije od *fork*() u slučaju kopiranja pri upisu. Korišćenjem *vfork*() roditeljski proces se suspenduje, a dete proces koristi adresni prostor roditelja. Budući da *vfork*() ne koristi kopiranje pri upisu, ako dete proces promeni bilo koju stranicu adresnog prostora roditelja, izmenjene stranice će biti vidljive roditelju nakon što se nastavi njegovo izvršavanje. Pošto se ne vrši kopiranje stranica, *vfork*() je izuzetno efikasna metoda kreiranja procesa.

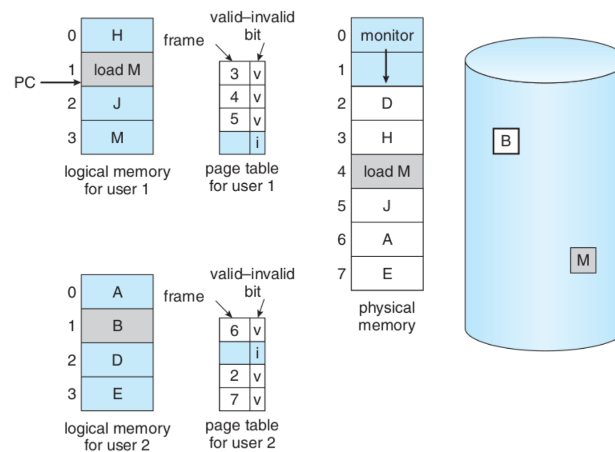
4 Zamena stranica (Page replacement)

U ranijoj analizi procenta postojećih *grešaka stranica* prilikom referenciranja memorije, pretpostavili smo da svaka referenciranje svake stranica dovodi do greške stranice najviše jednom, i to u trenutku kada se ona prvi put referencira. Međutim, ovakav zaključak nije nužno tačan. Ako proces koji sadrži deset stranica zapravo koristi samo polovinu njih, mehanizam stranica na zahtev štedi U/I operacije potrebne za učitavanje dodatnih pet stranica koje se nikada ne koriste. Kao rezultat, mogli bismo da povećamo stepen multi-programiranja, izvršavanjem dvostruko više procesa. Dakle, da imamo četrdeset okvira, mogli bismo da pokrenemo osam procesa, umesto četiri koji bi se maksimalno mogli pokrenuti ako svaki od njih zahteva deset okvira (od kojih pet nikada nije korišćeno).

Ako povećamo stepen multi-programiranja, prekomerno alociramo memoriju. Ako pokrenemo šest procesa, od kojih je svaki veličine deset stranica, ali zapravo koristi samo pet stranica, imamo veću CPU iskorišćenost i propusnost procesora, uz deset okvira koji se mogu iskoristiti. Moguće je, međutim, da će svaki od ovih procesa u datoj situaciji možda odjednom pokušati da koristi svih deset svojih stranica, što će rezultirati potrebom za šezdeset okvira memorije kad je na raspolaganju samo četrdeset njih.

Dodatno, uzmite u obzir da se sistemska memorija ne koristi samo za smeštanje stranica korisničkih programa. U/I baferi takođe troše znatnu količinu memorije, koji mogu znatno zakomplikovati rad algoritama za raspodelu memorije. Odlučiti koliko memorije dodeliti U/I, a koliko stranicama korisničkih programa značajan je izazov. Neki sistemi dodeljuju fiksni procenat memorije za U/I bafere, dok drugi omogućavaju da se i korisnički procesi i U/I podsistem ravnopravno nadmeću za sistemsku memoriju.

Prekomerna alokacija memorije se manifestuje na sledeći način. Dok se korisnički proces izvršava, dolazi do greške stranice. Operativni sistem određuje gde se na disku nalazi željena stranica, ali tada utvrđuje da u listi slobodnih



Slika 9: Potreba za zamenom stranice

okvira nema slobodnih okvira jer je čitava memorija je u upotrebi (slika 9).

Operativni sistem u ovom trenutku ima nekoliko opcija. Mogao bi da terminira korisnički proces. Međutim, upotreba stranica na zahtev je pokušaj operativnog sistema da poboljša iskorišćenost i propusni opseg računarskog sistema. Korisnici ne bi trebali biti svesni da se njihovi procesi izvršavaju u kontekstu stranica, a stranica bi trebala biti logično transparentna za korisnika. Dakle, ova opcija nije najbolji izbor.

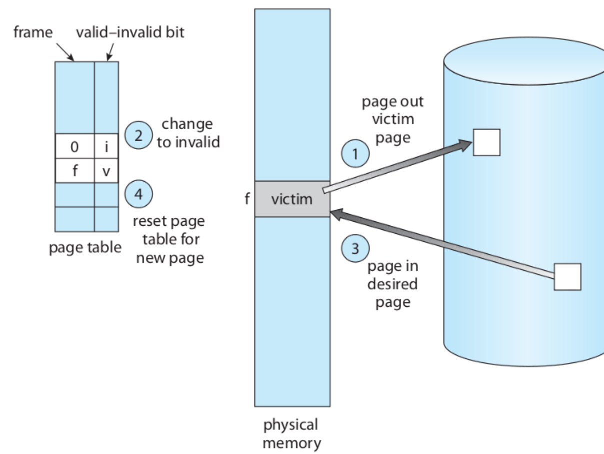
Operativni sistem bi umesto toga mogao da proces zameni iz memorije (*swap out*), oslobađajući sve njegove okvire i na taj način smanji stepen multiprogramiranja. Ova opcija je dobra u određenim okolnostima i razmatraćemo je kasnije, a ovde ćemo posmatrati najčešće rešenje: zamena stranice.

4.1 Osnovna zamena stranice

Ako nijedan okvir memorije nije slobodan, pronalazimo onaj koji se trenutno ne koristi i oslobađamo ga. Okvir možemo osloboditi tako da prepišemo njegov sadržaj u *prostor za zamenu* (*swap space*) i ažuriramo tabelu stranica (i sve ostale tabele) da označimo kako ta stranica više nije u memoriji (slika 10).

Sada možemo da koristimo oslobođeni okvir da smestimo stranicu čije referenciranje je dovelo do greške stranice datog procesa. Rutinu za obradu greške stranice ćemo promeniti tako da ona uključuje ovakvu zamenu stranice:

1. Pronađi lokaciju željene stranice na disku;
2. Pronađi slobodni okvir:
   (a) Ako postoji slobodni okvir, koristi ga.
   (b) Ako nema slobodnog okvira, koristi algoritam zamene stranice kako bi odabrao takav okvir (*okvir žrtvu* - eng. *victim frame*)



Slika 10: Zamena stranice

(c) Upiši okvir žrtve na disk, a u skladu sa tim promeni tabelu stranica i okvira

3. Učitaj željenu stranicu u novo-oslobođeni okvir, a promeni tabelu stranica i okvira;

4. Nastavi korisnički proces sa mesta gde se dogodila greška stranice.

Imajte na umu da su, ukoliko nijedan okvir nije slobodan, potrebna dva prenosa stranica (jedan iz memorije i jedan u memoriju). Ova situacija efektivno udvostručuje vreme servisiranja greške stranice i u skladu s tim povećava efektivno vreme pristupa.

Možemo smanjiti ove dodatne troškove korišćenjem bita modifikacije (zove se i *dirty bit*). Kada se koristi ova šema, svaka stranica ili okvir ima bit modifikacije dodeljen njoj u hardveru. Bit modifikacije date stranice hardver postavlja na 1 svaki put kad se upiše bilo koji bajt na toj stranici, što ukazuje da je stranica modifikovana. Kada odaberemo stranicu za zamenu, ispitujemo njen bit modifikacije. Ako je bit postavljen, znamo da je stranica modifikovana od momenta kada je učitana sa diska. U ovom slučaju moramo upisati stranicu na disk. Međutim, ako bit za modifikaciju nije postavljen, stranica nije modifikovana od kada je učitana u memoriju. U ovom slučaju ne moramo da upišemo stranicu na disk jer je ona već tamo. Ova tehnika se takođe odnosi i na stranice koje mogu samo da se čitaju (na primer, stranice koje sadrže izvršni kod). Takve stranice se ne mogu menjati, pa se njihov sadržaj može samo prepisati novim po potrebi. Ova šema može značajno da smanji vreme potrebno za obradu greške stranice, jer smanjuje U/I vreme za pola, ako stranica nije izmenjena.

Zamena stranice je osnovni koncept kod stranica za zahtev. On zapravo vrši konačno razdvajanje između logičke memorije i fizičke memorije. Ovim mehanizmom može se obezbediti ogromna virtualna memorija za programere, dok u

sistemu ima znatno manje fizičke memorije. Bez stranica na zahtev, korisničke adrese su mapirane u fizičke adrese, a ova dva skupa adresa mogu biti različiti. Međutim, sve stranice procesa moraju se nalaziti u fizičkoj memoriji. Kod stranica na zahtev, veličina logičkog adresnog prostora više nije ograničena fizičkom memorijom. Ako imamo korisnički proces od dvadeset stranica, možemo ga izvršavati u sistemu sa deset okvira jednostavno korišćenjem stranica na zahtev i algoritma zamene stranica kako bismo pronalašli slobodan okvira kad god je to potrebno. Ako se stranica koja je modifikovana treba zameniti, njen se sadržaj kopira na disk. Kasnije referenciranje na tu stranicu dovešće do greške stranice. Tada će se stranica vratiti u memoriju, a možda usput zameniti neku drugu stranicu datog procesa.

Dakle, da bismo implementirali stranice na zahtev, moramo rešiti dva osnovna problema: moramo razviti algoritam za raspoređivanje okvira i algoritam zamene stranica. To jest, ako imamo više procesa u memoriji, moramo odlučiti koliko okvira dodeliti svakom procesu (algoritam za raspoređivanje okvira), a kada je potrebna zamena stranice, moramo odabrati okvire koje treba zameniti (algoritam zamene stranica). Dizajn odgovarajućih algoritama za rešavanje ovih problema je važan zadatak jer su U/I operacije sa diskom izuzetno skupe u pogledu vremena. Čak i mala poboljšanja ovih algoritama donose velike dobitke u performansama sistema.

Postoji mnogo različitih algoritama zamene stranica. Svaki operativni sistem uglavnom koristi svoju šemu zamene. Kako odabrati najbolji algoritam zamene? Generalno, želimo da imamo onaj algoritam koji obezbeđuje najnižu stopu grešaka stranica.

Algoritam procenjujemo tako da ga izvršavamo na određenom nizu referenci memorije i izračunavamo broj grešaka stranica. Niz referenci u memoriji naziva se *referentnim nizom*. Možemo da generišemo referentne nizove veštački (na primer, korišćenjem generatora slučajnih brojeva) ili možemo da pronađemo određeni sistem i zabeležimo adresu svake reference memorije.

Druga metoda proizvodi veliki broj podataka (~1 milion adresa u sekundi), a da bismo smanjili broj podataka sa kojima radimo, koristimo dve činjenice. Prvo, za određenu veličinu stranice (a veličina stranice obično je određena i fiksirana bilo hardverom ili od strane sistema), uzimamo u obzir samo broj stranice, a ne celu adresu. Drugo, ako imamo referencu na stranicu p, tada svaka referenca na stranicu p koja sledi odmah nakon nje neće uzrokovati grešku stranice. Stranica p će biti u memoriji nakon prve reference, tako da sukcesivne reference na nju neće dovesti do greške stranice.

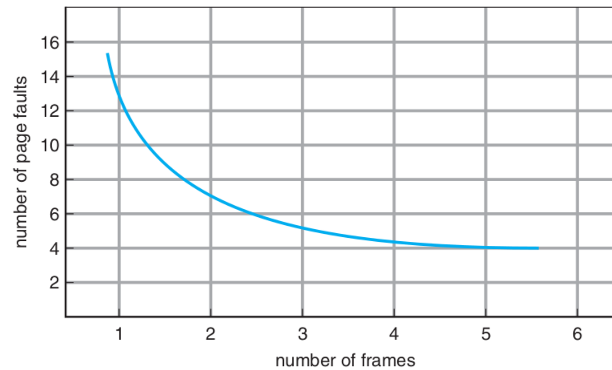
Na primer, ako pratimo određeni proces, možemo snimiti sledeću sekvencu adresa:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

Za 100 bajtova po stranici, ovaj niz se svodi na sledeći referentni niz:

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Da bismo odredili broj grešaka stranica za određeni referentni niz i algoritam zamene stranice, takođe moramo znati i broj dostupnih (slobodnih) okvira stranica. Naravno, kako se povećava broj dostupnih okvira, smanjuje se i broj



Slika 11: Zavisnost grešaka stranica od broja okvira memorije u sistemu

grešaka na stranici. Na primer, za prethodno razmatrani referentni niz, ukoliko bismo imali tri ili više okvira, imali bismo samo tri greške stranice (po jednu grešku za prvo referenciranje na svaku stranicu). Nasuprot tome, sa samo jednim okvirom na raspolaganju, imali bismo zamenu sa svakom referencom, što rezultuje sa jedanaest grešaka stranica. Uopšte, očekujemo krivu zavisnosti grešaka stranica od broja dostupnih okvira poput one prikazane na slici 11.

Kako se broj okvira povećava, broj grešaka stranica opada na neki minimalan nivo. Naravno, proširenjem fizičke memorije povećava se broj dostupnih okvira.

U nastavku analiziraćemo nekoliko algoritama za zamenu stranica. Pri tome koristimo referentni niz

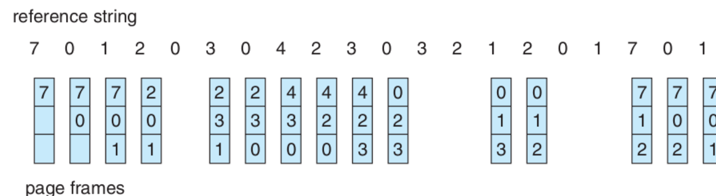
7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

i memoriju sa sa tri okvira.

4.2 FIFO algoritam zamene stranica

Najjednostavniji algoritam za zamenu stranica je FIFO algoritam. FIFO algoritam za zamenu dodeljuje svakoj stranicom vreme kada je ta stranica učitana u memoriju. Kada se stranica mora zameniti, bira se najstarija učitana stranica. Pri tome, nije neophodno beležiti vreme kada je stranica učitana. Možemo koristiti FIFO red (otuda naziv za algoritam) u kojem će se smeštati brojevi svih stranica učitanih u memoriju. Kada se vrši zamena uzimamo stranicu sa početka reda, a kada se stranica učitava u memoriju, ubacujemo je na kraj reda.

Za naš referentni niz, tri okvira su u početku prazna. Prve tri reference (7, 0, 1) uzrokuju greške stranica, nakon čega se stranice 7, 0 i 1 učitavaju u ove prazne okvire. Sledeća referenca (2) zamenjuje stranicu 7, jer je stranica 7 prva učitana. Pošto je 0 sledeća referenca i 0 je već u memoriji, za ovu referencu nemamo grešku stranice. Prva referenca na 3 rezultuje zamenom stranice 0, jer je ona sada prva na redu za zamenu. Zbog ove zamene, sledeća referenca na 0 će dovesti do greške stranice. Stranica 1 se tada zamenjuje stranom 0, a ovaj postupak se nastavlja kao što je prikazano na slici 12. Svaki put kada dođe do



Slika 12: FIFO algoritam zamene stranica

greške stranice, prikazano je koje su stranice smeštene u tri dostupna okvira memorije. Ukupno će biti petnaest grešaka stranica za dati referentni niz.

FIFO algoritam zamene stranica lak je za kako za razumevanje tako i za implementaciju. Međutim, njegove performanse nisu uvek dobre. S jedne strane, zamenjena stranica može biti modul za inicijalizaciju koji je korišćen pri podizanju sistema i više nije potreban. S druge strane, mogao bi sadržati često korišćenu promenljivu koja se inicijalizovala rano i u stalnoj je upotrebi.

Važno je primetiti da, čak i ako odaberemo za zamenu stranicu koja je u aktivnoj upotrebi, sve i dalje ispravno radi. Nakon što aktivnu stranicu zamenimo novom, gotovo odmah dolazi do greške prilikom referenciranja aktivne stranice. Neka druga stranica mora biti zamenjena kako bi se aktivna stranica vratila u memoriju. Stoga, loš izbor zamene povećava broj grešaka stranica i usporava izvršavanje procesa, ali ne rezultuje pogrešnim izvršavanjem sistema generalno.

Da bismo ilustrovali probleme koji su mogući sa FIFO algoritmom zamene stranice, posmatramo sledeći referentni niz:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

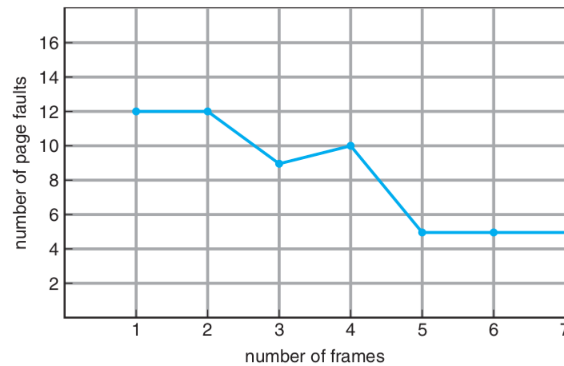
Na slici 13 prikazana je kriva grešaka stranica za dati referentni niz u zavisnosti od broj dostupnih okvira. Primetite da je broj grešaka stranica za četiri okvira (deset) veći od broja grešaka stranica za tri okvira (devet)! Ovaj najneočekivaniji rezultat poznat je kao *Beladijeva anomalija* (eng. *Belady anomaly*). Za određene algoritme zamene stranice stopa grešaka stranica može se povećati kako se broj dostupnih okvira povećava. Očekivali bismo da će dodeljivanje više memorije nekom procesu poboljšati njegove performanse, ali očigledno je da ta pretpostavka nije uvek tačna.

4.3 Optimalna zamena stranica

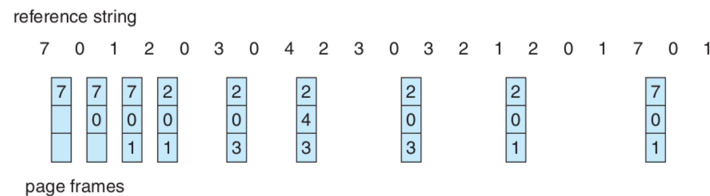
Jedan rezultat otkrića Beladijeve anomalije bila je potraga za optimalnim algoritmom zamene stranice. Takav algoritam obezbedio bi najnižu stopu grešaka stranica od svih algoritama i ni za redan referentni niz neće pokazati Beladijevu anomaliju. Takav algoritam postoji i nazvan je OPT (ili MIN). OPT algoritam je jednostavan i kaže:

Zameni stranicu koja se neće koristiti najduže vremena

Upotreba ovog algoritma zamene stranica garantuje najmanju moguću stopu greške stranica za fiksni broj okvira. Na primer, u slučaju našeg referentnog



Slika 13: Beladijeva anomalija

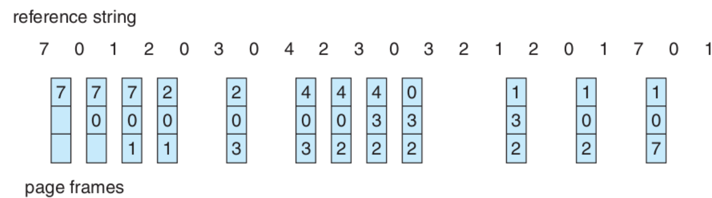


Slika 14: Optimalna zamena stranica

referenci, optimalni algoritam zamene stranica dao bi devet grešaka stranica, kao što je prikazano na slici 14.

Prve tri reference uzrokuju greškama i kao rezultat učitaće se u tri prazna okvira. Referenca na stranicu 2 zamenjuje stranicu 7, jer se stranica 7 neće koristiti sve do reference 18, dok će stranica 0 biti upotrebljena u referenci 5, a stranica 1 u referenci 14. Referenca na stranicu 3 zamenjuje stranicu 1, jer će stranica 1 biti poslednja od tri stranice među okvirima koja će biti ponovo referencirana. Sa samo devet grešaka stranica, optimalna zamena je mnogo bolja od algoritma FIFO, koji na istom nizu daje petnaest grešaka. (Ako zanemarimo prve tri, koja su neminovne kod svakog algoritma, optimalna zamena je dvostruko bolja od FIFO zamene). Nijedan algoritam zamene ne može da prođe kroz ovaj referentni niz sa tri dostupna okvira sa manje od devet grešaka.

Nažalost, optimalni algoritam zamene stranica je teško implementirati, jer zahteva poznavanje budućeg referentnog niza. Nailazili smo na sličnu situaciju kod SJF algoritma za raspoređibanje procesora. Kao rezultat toga, optimalni algoritam koristi se uglavnom kao referentni. Na primer, bilo bi korisno znati da, iako dati algoritam nije optimalan, u najgorem slučaju je lošiji 12,3 procenta od optimalnog, a u proseku 4,7 procenata.



Slika 15: LRU zamena stranica

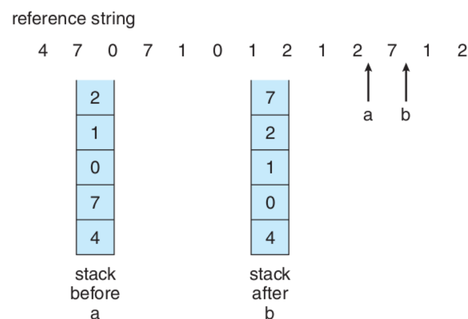
4.4 LRU zamena stranica

Ako optimalni algoritam nije moguće implementirati, postavlja se pitanje da li postoji aproksimacija optimalnog algoritma? Ključna razlika između FIFO i OPT algoritama (osim toga što jedan gleda unazad a drugi u napred u pogledu vremena) je da FIFO algoritam koristi vreme učitavanja stranice u memoriju, dok OPT algoritam koristi vreme kada se stranica *koristi*. Ako koristimo nedavnu prošlost kao aproksimaciju bliske budućnosti, tada možemo zameniti stranicu koja se nije koristila najduže vremena. Ovaj pristup je poznat kao *algoritam najduže nekorišćene stranice* (*Least Recently Used* - LRU). LRU zamena dodeljuje svakoj stranici vreme poslednje upotrebe te stranice. Kada se stranica mora zameniti, LRU bira stranicu koja se najduže ne koristi. O ovoj strategiji možemo razmišljati kao o optimalnom algoritmu zamene stranice koji gleda unatrag, a ne prema napred, u pogledu vremena. Interesantno je da, za S^R koji je invertovani niz od referentnog niza S, stopa greške stranice za OPT algoritam na S jednaka je stopi greške stranice za OPT algoritam na S^R. Slično tome, stopa greške stranica za LRU algoritam na S jednaka je stopi greške stranica za LRU algoritam na S^R.

Rezultat zamene korišćenjem LRU na našem primeru referentnog niza prikazan je na slici 15. LRU algoritam generiše dvanaest grešaka stranica. Pri tome, prvih pet grešaka su iste kao u slučaju algoritma za optimalnu zamenu. Kada se, ipak, dogodi referenca na stranicu 4, LRU algoritam vidi da je od tri okvira u memoriji stranica 2 najduže korišćena. Stoga LRU algoritam zamenjuje stranicu 2, ne znajući da će se stranica 2 koristiti odmah nakon toga. Kad potom dođe do greške za stranicu 2, algoritam LRU zamenjuje stranicu 3, jer je sada ona najmanje nedavno korišćena od tri stranice u memoriji. Uprkos očiglednim problemima, zamena korišćenjem LRU algoritma sa dvanaest grešaka mnogo je bolja nego zamena korišćenjem FIFO algoritma sa petnaest.

LRU algoritam se često koristi kao algoritam za zamenu stranica i smatra se dobrim. Glavni problem je kako implementirati zamenu na LRU način. Algoritam zamene stranice u maniru LRU može zahtevati značajnu pomoć hardvera, jer se problem svodi na to kako odrediti redosled za okvire u skladu sa trenutkom poslednje upotrebe. Dve implementacije su moguće:

1. Brojači. U najjednostavnijem slučaju, svakom unosu u tabelu stranica pridružujemo polje *vreme korišćenja*, dok CPU-u dodeljujemo logički brojač



Slika 16: LRU implementacija korišćenjem steka

koji se uvećava kod svakog referenciranja memorije. Kad god se referencira na neku stranicu, sadržaj brojača se kopira u polje *vreme korišćenja* kod unosa u tabeli stranica za datu stranicu. Na ovaj način uvek imamo „vreme" poslednje reference na svaku stranicu, a zamenjujemo stranicu sa *najmanjom* vrednošću vremena. Ova šema zahteva pretragu tabele stranica radi pronalaženja takve LRU stranice i upis u memoriju (u polje *vreme korišćenja* u tabeli stranica) za svaki pristup memoriji. Vremena korišćenja se, takođe, moraju čuvati prilikom zamene tabele stranica (zbog zamene konteksta i raspoređivanja CPU-a). Dodatno, mora se uzeti u obzir i eventualno prekoračenje brojača.

2. Stek. Drugi pristup zamene LRU stranice svodi se na održavanje steka na koji se upisuju brojevi stranica. Kad god se stranica referencira, ona se uklanja iz steka i stavlja se na njegov vrh. Na ovaj način, najskorije referencirana stranica je uvek na vrhu steka, a najduže nekorišćena stranica uvek je na dnu (slika 16). Budući da unosi moraju biti uklanjani iz sredine steka, najbolje je implementirati ovaj pristup pomoću dvostruko povezane liste s pokazivačima na početak i kraj liste. Uklanjanje stranice i stavljanje na vrh steka zahteva promenu šest pokazivača u najgorem slučaju. Svako ažuriranje u smislu promene elemenata steka je malo skuplje, ali se svakako ne mora tražiti stranica za zamenu kada bude trebalo, jer pokazivač na kraj steka uvek pokazuje na LRU stranicu. Ovaj pristup je posebno pogodan za softversku implementaciju ili implementaciju LRU algoritma u okviru mikrokoda.

Kao i optimalna zamena, zamena korišćenjem LRU ne pokazuje probleme u vezi sa Beladijevom anomalijom. Oba algoritma pripadaju specifičnoj klasi algoritama za zamenu stranica, koja se naziva *stek algoritmi*. Ne postoji referentni niz koji će dovesti da stek algoritmi pokažu Beladijevu anomaliju. Algoritam steka je algoritam za koji se može pokazati da je skup stranica u memoriji za n okvira uvek podskup skupa stranica koji bi bio u memoriji sa $n+1$ okvira. Kod LRU algoritma, skup stranica koje su u memoriji uvek će biti n stranica sa

najskorijim referencama. Ako se broj dostupnih okvira poveća, ovih n stranica će i dalje biti najskorije referencirane, tako da će ostati u memoriji.

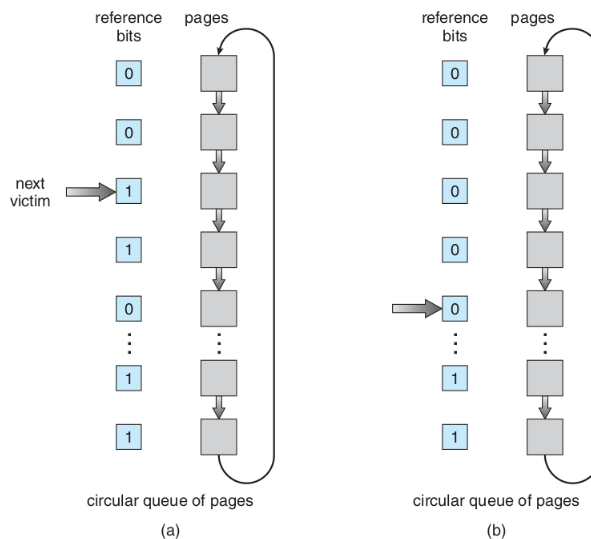
Ipak, treba imati na umu da je svaka gore opisana implementacija LRU algoritma, iako moguća, izuzetno skupa u pogledu vremena i resursa. Ažuriranja polja *vreme korišćenja* ili steka moraju se izvršiti kod svake reference memorije. Ako bismo koristili prekid za svaku referencu, kako bismo softveru omogućili da ažurira takve strukture podataka, svaka referenca memorije bi se usporila faktorom od najmanje deset, a samim tim bi se približno usporio i korisnički svaki. Malo je sistema koji bi mogli tolerisati takvu implementaciju sistema za upravljanje memorijom.

4.5 Aproksimacije LRU algoritma zamene stranica

Malo računarskih sistema pruža dovoljno hardverske podrške neophodne za implementaciju zamene stranice korišćenjem LRU. Ipak, mnogi sistemi pružaju određenu pomoć u obliku *bita reference* (eng. *reference bit*). Bit referenci za datu stranicu postavlja hardver kada god se ta stranica referencira (bilo da je u pitanju čitanje ili upis, bilo kojeg bajta u sklopu stranice). Bitovi referenci dodeljeni su svakom unosu u tabeli stranica. U početku se svi bitovi brišu (postavljaju na 0) od strane operativnog sistema. Kako se korisnički proces izvršava, hardver postavlja bit referenci (povezan sa svakom referenciranom stranicom) na 1. Nakon nekog vremena, možemo utvrditi koje su stranice korišćene, a koje nisu korišćene ispitivanjem bitova referenci, iako ne znamo tačan redosled referenciranja stranica. Ove informacije su osnova za mnoge algoritme zamene stranica koji aproksimiraju LRU zamenu.

4.5.1 Algoritam višestrukih bita reference

Dodatne informacije o redosledu referenciranja dobijamo tako što proveravamo bite referenci u regularnim vremenskim intervalima i beležimo njihove vrednosti. Kao primer, koristimo 8-bitni podatak asociran svakoj stranici i prekide koji se dešavaju, na primer, svakih 100ms. Svaki put kada se desi prekid, kontrola se prepušta operativnom sistemu a operativni sistem upisuje na poziciju najvišeg bita u 8-bitnoj reči bit reference, nakon što prethodni sadržaj reči pomeri u desno za jedno mesto, pri čemu se sadržaj na najnižem bitu odbacuje. Tada, ove 8-bitne reči sadrže istoriju korišćenja stranice na koju se odnose, tokom osam poslednjih vremenskih intervala. Na primer, ako za datu stranicu ova reč ima vrednost 00000000, znamo da ona nije referencirana u osam prethodnih intervala. Sa druge strane, stranica koja je barem jednom referencirana tokom svakog vremenskog intervala imaće vrednost 11111111. Stranica sa istorijom 11000100 je korišćena skorije od stranice sa istorijom 01110111. Ako tretiramo ove vrednosti kao neoznačene cele brojeve, stranica sa najmanjom vrednosti će biti LRU stranica i može se koristiti za zamenu. Naravno, vrednosti ne moraju biti jedinstvene, a u slučaju da imamo više stranica sa identičnim istorijatom bita reference, možemo zameniti svaku od njih ili jednostavno odabrati među njima u FIFO maniru.



Slika 17: Algoritam zamene stranica sa dvostrukom šansom

Broj bita koji se koristi za praćenje istorije bita reference, naravno, može da se menja. Odabiramo ga, u skladu sa dostupnim hardverom, tako da ažuriranje višestrukih bita reference može najbrže da se sprovede. U jednom ekstremnom slučaju, broj može biti sveden na 0, što rezultuje praćenjem samo bita reference. Ovaj algoritam naziva se *algoritam zamene stranica sa dvostrukom šansom* (eng. *second-chance page-replacement algorithm*).

4.5.2 Algoritam zamene stranica sa dvostrukom šansom

Osnova algoritma sa dvostrukim prolazom je FIFO algoritam zamene. Međutim, kada je data stranica odabrana, proverava se njen bit reference. Ako je vrednost bita reference 0, nastavljamo sa zamenom te stranice. Ali ako je bit reference postavljen na 1, dajemo toj stranici priliku da opstane i nastavljamo do sledeće stranice u FIFO redu. Kada se data stranica preskoči, njen bit reference je postavljen na 0, a *vreme pristizanja* za tu stranicu je postavljeno na trenutno vreme. Kao rezultat, stranica kojoj je data druga šansa, neće biti zamenjena sve dok se sve ostale stranice ne zamene (ili je svim ostalima data druga šansa, takođe). Dodatno, ako se stranica koristi dovoljno često tako da njen bit reference bude uvek postavljen na 1, stranica nikad neće biti zamenjena.

Jedan način da se implementira ovaj algoritam zamene stranica sa dvostrukom šansom jeste korišćenjem cirkularnog reda. Pokazivač pokazuje na stranicu koja će sledeća biti zamenjena. U momentu kada je potreban dostupan okvir, pokazivač se kreće sve dok ne pronađe stranicu kod koje je bit reference jednak 0. Dok prolazi kroz red, usput postavlja sve bite referenci na 0, a kada se pronađe okvir žrtva, nova stranica se umeće na njeno mesto. U najgorem slučaju,

kada su svi biti reference setovani, pointer prolazi kroz ceo red, dajući drugu šansu svakom elementu reda. Obzirom na to da čisti bite reference pre nego što odabira stranicu koja će se zameniti, biće odabrana stranica sa najstarijim vremenom pristizanja. U ovom scenariju (kada su svi biti referenci postavljeni na 1), zamena stranica sa dvostrukom šansom degenerisana je u FIFO zamenu.

4.5.3 Poboljšan algoritam zamene stranica sa dvostrukom šansom

Možemo poboljšati prethodni algoritam praćenjem bita modifikacije zajedno sa bitom reference. Ovaj uređeni par (bit reference, bit modifikacija) definiše sledeće klase stranica

1. (0, 0) stanica koja nije skoro ni referencirana ni modifikovana - idealna za zamenu
2. (0, 1) nije skoro korišćena (referencirana) ali jeste modifikovana - nije idealan kandidat za zamenu, obzirom na to da će morati biti upisana na disk ukoliko je ona odabrana za zamenu
3. (1, 0) skoro korišćena ali nije modifikovana - vrlo verovatno će biti korišćena uskoro
4. (1, 1) skoro korišćena i modifikovana - verovatno će biti korišćena ponovo uskoro, a moraće biti kopirana na disk pre nego što bude zamenjena

Svaka stranica se nalazi u jednoj od klasa od gore. Kada je potrebno vršiti zamenu, koristimo isti mehanizam kao kod prethodno opisanog algoritma, ali umesto da proveravamo da li je bit reference postavljen na 1, proveravamo kojoj klasi stranica pripada ova stranica. Tada ćemo:

1. proći kroz cirkularni red u potrazi za stranicom iz klase (0,0). Ako je nađemo, nju odabiramo za zamenu;
2. ako je prvi prolaz kroz petlju neuspešan, prolazimo drugi put kroz petlju u potrazi za stranicom iz klase (0,1). U ovom prolasku, svim stranicama koje obiđemo stavljamo bit referenci na 0;
3. ako je i drugi prolaz neuspešan, vraćamo se na korak 1 (eventualno 2 posle njega), znajući da će sada jedan od ta dva biti uspešan.

U najgorem slučaju četiri puta ćemo obići red u potrazi za najboljom stranicom za zamenu. Osnovna prednost ovog algoritma u odnosu na jednostavniju verziju algoritma jeste ta da se ovde prioritetno biraju za zamenu stranice koje nisu modifikovane, kako bismo smanjili broj dugotrajnih U/I operacija.

4.6 Algoritmi za baferovanje stranica

Dodatne procedure se koriste zajedno sa algoritmima za zamenu stranica. Na primer, u određenim slučajevima, sistemi održavaju *skup slobodnih okvira*.

Kada dođe do greške stranice, okvir žrtva se nalazi kao i inače. Međutim, željena stranica se učitava u okvir iz *skupa slobodnih okvira* pre nego što se okvir žrtva prepiše na disk. Ova procedura omogućava procesu da se ponovo pokrene što je pre moguće, bez čekanja da se oslobodi okvir u koji se može upisati potrebna stranica. Kada se, vremenom, sadržaj okvira žrtve prepiše u sekundarnu memoriju, taj okvir se dodaje skupu slobodnih okvira.

Proširenje ove ideje svodi se na održavanje liste modifikovanih stranica. Svaki put kada je hardver za straničenje besposlen, modifikovane stranice se odabiraju i upisuju na disk. Dodatno, bit modifikacije je ažuriran i postavljen na 0. Ovaj metod povećava verovatnoću da će stranica moći samo da sa prepiše novim sadržajem kada bude odabrana za zamenu, bez dodatnog prepisivanja sadržaja stranice na disk.

Još jedna modifikacija se odnosi na održavanje skupa slobodnih okvira, uz dodatak da se pamti koja stranica je bila upisana u kom okviru. Obzirom na to da se sadržaj okvira neće menjati nakon što je okvir upisan na disk, stara stranica može biti direktno iskorišćena iz skupa slobodnih okvira, kada je potrebno, pod uslovom da okvir nije menjan u međuvremenu. Ovakav pristup eliminiše potrebu za dodatnom U/I operacijom na disku: kada dođe do greške stranice, prvo proveravamo da li se tražena stranica nalazi u skupu slobodnih okvira. Ako nije moramo odabrati okvir za zamenu i u njega učitati stranicu sa diska.

5 Alokacija memorijskih okvira

Kako alocirati fiksan broj dostupnih memorijskih okvira između višestrukih procesa kojima su oni potrebni? Ako imamo 93 okvira i dva procesa, koliko treba alocirati prvom, a koliko drugom? Najjednostavniji slučaj je sistem sa jednim korisnikom. Razmatramo ovakav sistem sa 128 kB memorije i stranicama od 1kB (128 okvira). Operativni sistem može zauzeti 35kB, što ostavlja 93 okvira korisničkom procesu. Kod stranica isključivo na zahtev, sva 93 okvira će inicijalno biti stavljeni u listu slobodnih okvira. Kada proces krene sa izvršavanjem, generisaće sekvencu grešaka stranica. Prve 93 greške stranice će dobiti raspoložive okvire iz liste slobodnih okvira, a kada se oni svi iskoriste, algoritam za zamenu stranica će biti korišćen kako bi se odredilo koji okvir od njih 93 da bude zamenjen 94-tim i tako dalje. Kada se proces terminira, 93 okvira se opet vraćaju u listu slobodnih okvira. Postoje brojne varijacije ove jednostavne strategije.

5.1 Minimalan broj okvira

Osim što ne možemo alocirati više memorijskih okvira od ukupnog broja postojećih u sistemu, postoji i minimalan broj okvira koji se moraju alocirati za dati proces. Jedan razlog za postojanje minimalnog broja okvira je u vezi sa performansama. Očigledno, kako se broj okvira dodeljenih procesu smanjuje, stopa grešaka stranica se uvećava, što usporava izvršavanje procesa. Dodatno,

kada se greška stranice desi pre nego što se završi instrukcija koja se izvršava, instrukcija se mora izvršavati ponovo od početka. Kao posledica, moramo imati dovoljno okvira kako bismo skladištili sve stranice koje bilo koja pojedinačna instrukcija referencira. Ovaj minimalan broj okvira je, samim tim, određen i definisan arhitekturom procesora. Suprotno ovome, maksimalan broj okvira je određen količinom dostupne fizičke memorije. Između ove dve vrednosti, alokacija okvira ostavlja brojne izbore.

5.2 Algoritmi za alokaciju okvira

Najjednostavniji način da se m okvira rasporedi na n procesa jeste da se svakom procesu dodeli jednak broj, m/n okvira (ako ignorišemo okvire koji se koriste od strane operativnog sistema). Na primer, ako postoji 93 okvira i pet procesa, svaki proces će dobiti 18 okvira. Tri preostala okvira mogu da se koriste u kao deo skupa slobodnih okvira. Ovakva shema je poznata kao *ravnopravna alokacija*.

Alternativa ovom pristupu jeste da se prepozna kako različiti procesi mogu imati drugačije zahteve za memorijom. Za sistem sa stranicama veličine 1kB, ako mali proces zahteva samo 10kB, a zahtevni 127kB i ta dva su jedini u sistemu sa 62 slobodna okvira, nema mnogo smisla dati svakom od njih po 31 okvir. Kako bi se rešio ovaj problem, koristi se proporcionalna alokacija, kod kojie se alocira dostupna memorija procesima u skladu sa njihovim potrebama. Ako je veličina virtualne memorije za proces p_i jednaka s_i i definišemo

$$S = \sum s_i$$

Tada, ako je ukupan broj dostupnih okvira m, alociraćemo a_i okvira procesu p_i, gde je a_i:

$$a_i = s_i/S \cdot m$$

Naravno, moramo podesiti svaki a_i da bude ceo broj, koji je veći od minimalnog broja okvira zahtevanog od strane datog skupa instrukcija, sa ukupnom sumom koja ne prelazi m.

Proporcionalnom alokacijom, 62 okvira iz primera od malopre bismo podelili tako što alociramo 4 okvira jednom i 57 okvira drugom procesu. Na ovaj način, oba procesa dele dostupne okvire u skladu sa svojim potrebama, umesto ravnopravne alokacije.

Kako kod ravnopravne, tako i kod proporcionalne alokacije, naravno, sama alokacija se može menjati u skladu sa stepenom multi-programiranja. Ako je stepen multi-programiranja povećan, svaki proces će izgubiti neke od okvira koje je koristio, kako bi se obezbedila potrebna memorija za novi proces. Suprotno, ako se stepen multi-programiranja smanji, okviri korišćeni od strane procesa koji se više ne izvršava biće raspoređeni među trenutno izvršavanim procesima.

Takođe, obe alokacije predstavljene iznad ravnomerno tretiraju procese visokog i niskog prioriteta. Međutim, u skladu sa definicijom, moguće je da želimo

više memorije alocirati procesu višeg prioriteta kako bismo ubrzali njegovo izvršavanje, nauštrb procesa niskog prioriteta. Jedno rešenje bi bilo da se koristi proporcionalna alokacija gde se kod proračuna alocirane memorije okvira ne koristi relativna veličina procesa, nego prioritet procesa ili kombinacija veličine i prioriteta.

5.3 Globalna i lokalna alokacija

Drugi važan faktor u načinu na koji se okviri dodeljuju različitim procesima je zamena stranica. Ukoliko imamo više procesa koji se takmiče ravnopravno za okvire memorije, algoritme zamene stranica možemo klasifikovati u dve kategorije: globalna zamena i lokalna zamena. Globalna zamena omogućava procesu da odabere okvir za zamenu iz skupa svih okvira memorije, čak i ako je taj okvir trenutno dodeljen nekom drugom procesu, to jest, jedan proces može preuzeti okvir od drugog procesa. Lokalna zamena zahteva da svaki proces odabira samo iz skupa sopstvenih dodeljenih okvira.

Na primer, razmotrite šemu alokacije okvira u kojoj dopuštamo procesima visokog prioriteta da uzimaju okvire za zamenu od procesa sa niskim prioritetom. Proces može odabrati okvir za zamenu iz sopstvenih okvira ili okvira bilo kojeg procesa nižeg prioriteta. Ovaj pristup omogućava procesu visokog prioriteta da uveća svoj skup dodeljenih okvira na štetu procesa sa nižim prioritetom.

Sa strategijom lokalne zamene, broj okvira dodeljenih nekom procesu se ne menja. Kod globalne zamene, može se dogoditi da proces odabira samo okvire dodeljene drugim procesima, povećavajući na taj način broj sebi dodeljenih okvira (uz pretpostavku da drugi procesi ne odabiraju za zamenu njegove okvire).

Jedan problem u vezi sa algoritmom globalne alokacije je što proces ne može kontrolisati sopstvenu stopu grešaka stranice. Skup stranica u memoriji za proces zavisi ne samo od načina zamene stranica tog procesa, već i od načina na koji drugi procesi to rade. Zbog toga se isti proces može izvršavati sasvim drugačije isključivo usled spoljnih faktora (na primer, jedno za izvršavanje je potrebno 0,5 sekundi, a 10,3 sekundi za sledeće izvršavanje). To nije slučaj kod algoritma lokalne alokacije. Kod lokalne zamene, na skup stranica u memoriji procesa utiče isključivo zamena stranica tog procesa. Međutim, lokalna zamena može da ometa izvršavanje procesa tako što mu ne daje na raspolaganje dodatne, nekorišćene stranice memorije iz sistema. Stoga, globalna zamena generalno vodi ka većoj propusnosti sistema i zbog toga je ona najčešće korišćena metoda.

5.4 Ne-uniforman pristup memoriji

Do sada smo pretpostavljali da je kompletna osnovna memorija homogena, ili barem da se njoj pristupa jednako. Na mnogim računarskim sistemima to nije slučaj. Često, u sistemima sa više CPU-a, određeni CPU može pristupiti nekim delovima glavne memorije brže nego drugima. Ove razlike u performansama uzrokovane su načinom na koji su CPU i memorija međusobno povezani

u sistemu. Takav sistem se često sastoji od nekoliko sistemskih ploča od kojih svaka sadrži više procesora i deo memorije. Sistemske ploče međusobno su povezane na različite načine (od sistemskih magistrala do mrežnih veza velike brzine). Kao što se može očekivati, procesori na određenoj ploči mogu pristupiti memoriji na toj ploči sa manjim kašnjenjem nego što mogu pristupiti memoriji na drugim pločama u sistemu. Sistemi u kojima se vremena pristupa memoriji značajno razlikuju, poznati su kao *sistemi sa ne-uniformnim pristupom memoriji* (eng. *Non-Uniform Memory Access*, NUMA) i bez izuzetka su sporiji od sistema u kojima se memorija i CPU nalaze na istoj matičnoj ploči.

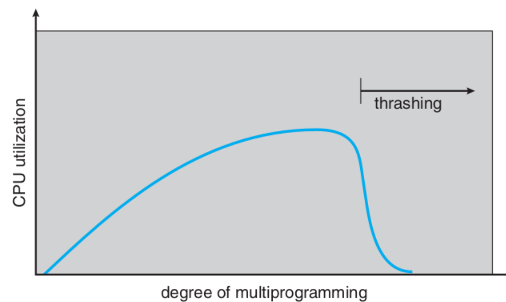
Odluke koji se okviri stranica čuvaju na kojim memorijskim lokacijama mogu značajno uticati na performanse u NUMA sistemima. Ako u takvom sistemu tretiramo memoriju kao homogenu, CPU može da čeka znatno duže prilikom pristupa memoriji nego u slučaju kada bismo NUMA arhitekturu uzeli u obzir kod algoritama za dodelu memorije. Slične izmene moraju se vršiti i u sistemu raspoređivanja procesora u NUMA okruženju. Cilj ovih modifikacija jeste da memorijski okviri budu dodeljeni „što je bliže moguće" procesoru na kome se izvršava proces. Definicija blizine u ovom smislu odnosi se na „minimalno kašnjenje", što obično znači na istoj sistemskoj ploči kao i CPU.

Algoritamske promene odnose se na to da planer prati poslednji CPU na kojem se izvršavao svaki proces. Ako planer pokušava da rasporedi svaki proces na prethodno korišćenom procesoru, a sistem za upravljanje memorijom da dodeli procesu okvire memorije koji se nalaze u blizini procesora na kojem je raspoređen proces, tada će doći do poboljšanja keš pogodaka i smanjenog vremena pristupa memoriji.

6 Preopterećenost virtualnog adresnog prostora (eng. *Thrashing*)

Ako broj okvira dodeljenih procesu niskog prioriteta padne ispod minimalnog broja koji zahteva arhitektura računara, moramo obustaviti njegovo izvršavanje. Nakon toga, neophodno je kopirati stranice korišćene od strane procesa na disk i nakon toga osloboditi sve njemu dodeljene okvire. Na ovaj način se vrši *zamena iz memorije*, odnosno *zamena u memoriju* od strane srednjeročnog planera, o čemu smo već pričali tokom predavanju o procesima.

Da bismo razumeli potencijalni problem sa kojim se srećemo, posmatrajmo bilo koji proces koji nema „dovoljno" okvira na raspolaganju. Ako proces nema dovoljno okvira kako bi se obezbedila podrška stranicama koje se aktivno koriste (na primer nova stranica referencirana od strane koda iz aktivne stranice), veoma brzo će doći do izuzetka greške stranice. U ovom trenutku moramo zameniti neku stranicu. Međutim, pošto su sve stranice tog procesa u aktivnoj upotrebi, mora odmah biti zamenjena stranica koja će procesu biti potrebna veoma uskoro. Kao rezultat, brzo ponovo dolazi do izuzetka greške stranice, i ponovo, i ponovo, zamenjujući pri tome stranice koje se moraju odmah vratiti u memoriju.



Slika 18: Preopterećenost

Ovakva aktivnost kod koje se generišu greške stranica i vrši prekomerna zamena stranica naziva se *preopterećenost*. Proces je *preopterećen* ako troši više vremena na straničenje nego što ga troši na izvršavanje.

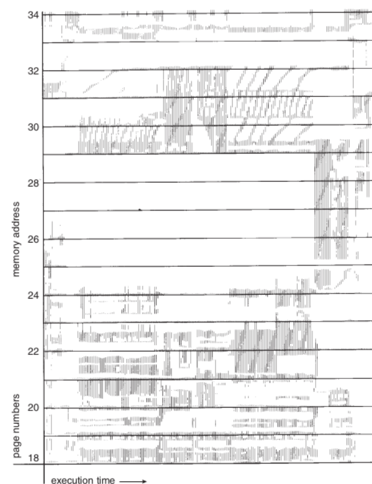
6.1 Uzrok preopterećenosti

Preopterećenost rezultira ozbiljnim problemima u vezi sa performansama sistema. Razmotrite sledeći scenario, koji se zasniva na stvarnom ponašanju prvobitnih sistema straničenja.

Operativni sistem nadgleda iskorišćenost procesora. Ako je iskorišćenost procesora previše niska, povećavamo stepen multi-programiranja uvođenjem novog procesa u sistem. Koristi se globalni algoritam zamene stranica i zamenjuje se stranica bez obzira kojem procesu pripada. Pretpostavimo sada da proces prelazi u sledeću fazu izvršavanja u kojoj mu treba više okvira memorije. Kao rezultat, proces počinje da generiše greške stranica i uzima okvire od drugih procesa. Ovi procesi, međutim, trebaju te stranice, pa tako i oni generišu greške stranica, preuzimajući okvire od ostalih procesa. Procesi koji generišu greške stranica moraju koristiti hardver za straničenje kako bi zamenili stranice *iz memorije* i *u memoriju*. Dok čekaju u redu za uređaj koji će izvršiti zamene, red spremnih procesa se prazni. Kao rezultat, dok procesi čekaju na korišćenje uređaja za straničenje, iskorišćenost CPU-a opada.

CPU planer primećuje ovu smanjenu upotrebu procesora i kao rezultat povećava dodatno stepen multi-programiranja. Novi proces pokušava započeti izvršavanje preuzimanjem okvira od već pokrenutih procesa, uzrokujući još više grešaka stranica i još duži red čekanja za uređaj straničenja. Kao rezultat toga, iskorišćenost procesora još više opada, a CPU planer pokušava još više da poveća stepen multi-programiranja. Došlo je do preopterećenosti, a propustnost sistema se značajno smanjila. Stopa grešaka stranica se strahovito uvećala. Kao rezultat, efektivno vreme pristupa memoriji se povećava. Koristen posao se ne izvršava, jer procesi troše svo vreme na zamene iz i u memoriju.

Ovaj fenomen je ilustrovan na slici 18, na kojoj je prikazana zavisnost iskorišćenosti procesora od stepenu multi-programiranja. Kako se stepen multi-



Slika 19: Lokaliteti u šablonima pristupa memoriji

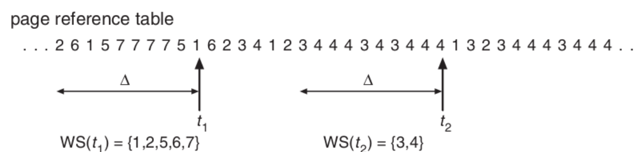
programiranja povećava, povećava se i upotreba CPU-a, iako sporije, dok se ne dostigne maksimum. Ako se stepen multi-programiranja još dodatno poveća, dolazi do preopterećenosti i iskorišćenost procesora naglo opada. U ovom trenutku, da bismo povećali iskorišćenost procesora i zaustavili preopterećenost, moramo *smanjiti* stepen multi-programiranja.

Efekat preopterećenosti možemo ograničiti korišćenjem algoritma *lokalne zamene*. Sa lokalnom zamenom, ako se jedan proces izvršava, on ne može preuzeti okvire od drugog procesa i prouzrokovati preopterećenost istog. Međutim, problem nije u potpunosti rešen na ovaj način. Ako su procesi preopterećeni, oni će većinu vremena provoditi u redu za uređaj koji obavlja straničenje. Prosečno vreme obrade izuzetka greške stranice će se povećati zbog dužeg prosečnog čekanja u redu uređaja za straničenje. Na taj način, efektivno vreme pristupa memoriji će se povećati čak i za proces koji nije preopterećen.

Da bismo sprečili preopterećenost, moramo dodeliti procesu onoliko okvira koliko mu treba.

Ali kako znati koliko okvira je njemu potrebno? Postoji nekoliko tehnika. Strategija *radnog skupa* prati koliko okvira se zapravo koristi u sistemu. Ovaj pristup definiše *lokalitetni model* izvršavanja procesa. *Lokalitetni model* određuje da, kako se proces izvršava, on prelazi sa *lokaliteta* na *lokalitet*. Lokalitet je, pri tome, skup stranica koje se aktivno koriste zajedno (slika 19).

Program se obično sastoji od više različitih lokaliteta, koji se mogu preklapati. Na primer, kada se funkcija pozove, ona definiše nov lokalitet. U ovom lokalitetu, memorijske reference se odnose na instrukcije u sklopu funkcije, njene lokalne promenljive i podskup globalnih promenljivih. Kada napustimo funkciju, proces napušta ovaj lokalitet, jer lokalne promenljive i instrukcije date funkcije više nisu u aktivnoj upotrebi. Na ovaj lokalitet ćemo se možda vratiti



Slika 20: Primer korišćenja modela radnog skupa

kasnije, ali trenutno nam više nije interesantan.

Dakle, vidimo da su lokaliteti definisani programskom strukturom i korišćenim strukturama podataka. Lokalitetni model tvrdi da će svi programi iskazati istu ili sličnu strukturu u pogledu referenciranja memorije. Naravno, sam model lokaliteta je generalni princip. On je fundamentalni model za keširanje - da su pristupi bilo kojoj vrsti podataka slučajni, tj. da ne važi princip lokalnosti, keširanje bi bilo potpuno beskorisno.

Pretpostavimo da smo dodelili dovoljno okvira procesu tako da može da se zadovolji njegov trenutni lokalitet. Greške stranica će se tada dešavati sve dok te stranice ne budu u memoriji, ali nakon toga neće dolaziti do novih grešaka stranica sve dok se ne promeni lokalitet. Ako ne dodelimo dovoljno okvira koji bi odgovarali veličini trenutnog lokaliteta, proces će vremenom postati preopterećen, jer ne može zadržati u memoriji sve stranice koje aktivno koristi.

6.2 Model radnog skupa (*Working-Set Model*)

Kao što je pomenuto, model radnog skupa zasnovan je na pretpostavci lokalnosti. Ovaj model koristi parametar Δ, za definisanje *prozora radnog skupa*. Ideja je da se prate reference stranica u okviru prozora radnog skupa. Stranice koje se nalaze među referencama stranica u okviru poslednjih Δ memorijskih referenci čine *radni skup* (slika 20).

Ako je stranica u aktivnoj upotrebi, ona će biti u radnom skupu. Ako se više ne koristi, biće uklonjena iz radnog skupa nakon Δ memorijskih referenci koje ne uključuju nju. Dakle, samim tim, radni skup je aproksimacija lokaliteta datog programa.

Na primer, obzirom na redosled memorijskih referenci prikazan na slici 20, ako je $\Delta = 10$ memorijskih referenci, tada je radni skup u trenutku t_1 $\{1, 2, 5, 6, 7\}$. U trenutku t_2, radni skup se promenio u $\{3, 4\}$.

Tačnost aproksimacije radnog skupa zavisi od izbora Δ. Ako je Δ premali, neće obuhvatiti celokupni lokalitet. Ako je, opet, Δ prevelik, može dovesti do preklapanja nekoliko lokaliteta. U ekstremnom slučaju, ako je Δ beskonačan, radni skup čini skup stranica koje se posećene tokom izvršavanja procesa.

Samim tim, najvažnija osobina radnog skupa je njegova veličina. Ako je izračunata veličinu radnog skupa, WSS_i, za svaki proces u sistemu, tada možemo posmatrati

$$D = \sum WSS_i$$

gde D onda predstavlja ukupnu potražnju za okvirima, od strane svih procesa. Svaki proces aktivno koristi stranice u svom radnom skupu. Dakle, za proces i treba WSS_i okvira. Ako je ukupna potražnja za okvirima veća od ukupnog broja raspoloživih okvira ($D > m$), doći će do preotерećenosti u sistemu, jer neki procesi neće imati dovoljno raspoloživih okvira. Kada je određen i postavljen parametar Δ, upotreba modela radnog skupa je jednostavna. Operativni sistem nadgleda radni skup svakog procesa i dodeljuje tom radnom skupu dovoljno okvira u skladu sa veličinom radnog skupa. Ako, nakon toga, ima u sistemu dovoljno raspoloživih slobodnih okvira, može se pokrenuti još jedan proces. Ako se zbir veličina radnih skupova povećava, prelazeći preko ukupnog broja dostupnih okvira, operativni sistem bira proces koji će suspendovati. Stranice tog procesa se zamenjuju iz memorije, a njegovi okviri se dodeljuju drugim procesima. Suspendovani proces može se ponovo pokrenuti kasnije.

Ova strategija korišćenja radnog skupa sprečava preopterećenost, uz zadržavanje stepena multi-programiranja najvišim mogućim. Na taj način se optimizuje iskorišćenost procesora.

Problem sa modelom radnog skupa odnosi se na praćenje i ažuriranje radnog skupa. Prozor radnog skupa je pokretni prozor, kao što smo videli. Kod svake memorijske reference, na jednom kraju prozora se pojavljuje nova referenca, a najstarija referenca ispada iz skupa na drugom kraju prozora. Stranica se nalazi u radnom skupu ako se referenca na nju nalazi bilo gde u prozoru radnog skupa.

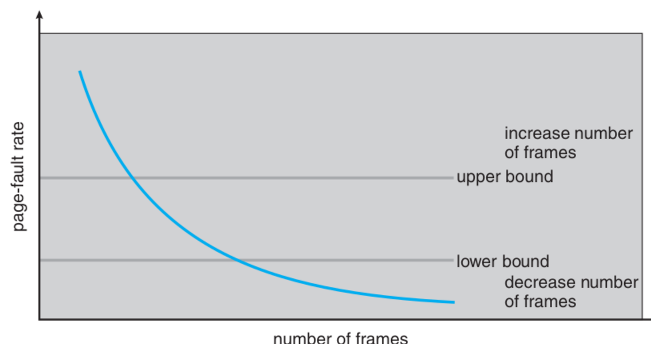
Model radnog skupa možemo približiti pomoću prekida tajmera sa fiksnim intervalom i bita reference. Na primer, pretpostavimo da je Δ jednako 10 000 referenci i da možemo izazvati prekid tajmera na svakih 5000 referenci. Kad dođe do prekida tajmera, kopiramo i brišemo bit reference za svaku stranicu. Prema tome, ako dođe do greške stranice, možemo ispitati trenutni bit reference i dva bita u memoriji kako bismo utvrdili da li je stranica korišćena u prethodnih 10 000 do 15 000 referenci. Ako je bila korišćena, barem jedan od ovih bita će biti postavljen na 1. Ako nije, biti će biti postavljeni na 0. Stranice sa barem jednim bitom postavljenim na 1 smatraće se elementima radnog skupa.

Treba imati na umu da ovaj mehanizam nije u potpunosti precizan, jer ne možemo reći kada se tačno, u intervalu od 5.000, dogodila referenca. Ovu nesigurnost možemo smanjiti povećanjem broja bita koji prate istoriju promena bita referenci ili skraćenjem perioda vremena i prekida tajmera (na primer, 10 bita i prekid na svakih 1.000 referenci). Međutim, u ovom slučaju cena obrade ovih dodatnih prekida biće znatno veća i možda neprihvatljiva za dati sistem.

6.3 Učestalost grešaka stranica

Model radnog skupa je uspešan i funkcionalan, ali deluje kao nepotrebno komplikovan mehanizam za kontrolu preopterećenosti. Strategija koja koristi *učestalost grešaka stranica* (eng. *Page-Fault Frequency*, PFF) ima direktniji pristup.

Specifični problem koji pokušavamo da rešimo jeste kako sprečiti preopterećenost. Preopterećenost se manifestuje visokom stopom grešaka stranica. Stoga, želimo da kontrolišemo učestalost grešaka stranica. Kad je ona previsoka, znamo



Slika 21: Učestalost greške stranica

da procesu treba više okvira. Suprotno tome, ako je stopa grešaka stranica preniska, tada proces možda ima previše okvira. Kao rezultat, možemo uspostaviti gornju i donju granicu na željenoj stopi greške stranice (slika 21).

Ako stvarna stopa grešaka stranica prelazi gornju granicu, dodeljujemo procesu još jedan okvir. Ako stopa greške stranice padne ispod donje granice, procesu oduzimamo okvir. Na taj način možemo direktno meriti i kontrolisati broj grešaka stranica kako bismo sprečili preopterećenost.

Kao i kod strategije radnog skupa, u datom trenutku ćemo možda morati suspendovati proces i zameniti ga *iz memorije* (*swap out*). Naime, ukoliko se stopa grešaka stranica uvećava, a slobodni okviri nisu dostupni, moramo odabrati neki proces, kopirati njegove stranice na disk, a njega suspendovati. Oslobođeni okviri se tada distribuiraju procesima sa visokom stopom grešaka stranica.

7 Memorijski mapirane datoteke

Razmatramo sekvencijalno čitanje datoteke na disku koristeći standardne sistemske pozive *open*(), *read*() *write*(). Svaki pristup datotekama zahteva sistemski poziv i pristup disku. Alternativno, možemo koristiti dosadašnje tehnike virtuelne memorije da bismo tretirali U/I na datotekama kao rutinske pristupe memoriji. Ovaj pristup, poznat kao *memorijsko mapiranje datoteka*, omogućava da se deo virtualnog adresnog prostora logički poveže sa datotekom. Kao što ćemo videti, to može dovesti do značajnog poboljšanja performansi.

7.1 Osnovni mehanizam

Memorijsko mapiranje datoteka vrši se preslikavanjem bloka diska na stranicu (ili stranice) u memoriji. Inicijalni pristup datoteci vrši se kao kod stranica na zahtev, što dovodi do greške stranice. Međutim, deo datoteke veličine stranice se čita iz fajl sistema u fizički okvir. Naknadna čitanja i upisivanja u

datoteku tretiraju se kao rutinski pristupi memoriji. Manipulisanje datotekama kroz memorijski podsistem, umesto korišćenja sistemskih poziva *read*() i *write*() pojednostavljuje i ubrzava pristup i upotrebu datoteke.

Pri tome, upisivanje u memorijski mapiranu datoteku ne zahteva nužno istovremeno (sinhrono) upisivanje u datoteku na disku. Neki sistemi ažuriraju fizičku datoteku u momentu kada operativni sistem periodično proverava da li je stranica u memoriji izmenjena. Svakako kada se datoteka zatvara, svi podaci mapirani u memoriju se zapisuju na disk i uklanjaju iz virtualne memorije procesa.

Neki operativni sistemi omogućavaju mapiranje memorije datoteka korišćenjem specifičnih sistemskih poziva (*mmap*), a koriste standardne sistemske pozive za vršenje svih ostalih U/I operacija na datotekama. Međutim, neki sistemi vrše memorijsko mapiranje datoteke bez obzira da li je datoteka predviđena za mapiranje u memoriji. Uzmimo Solaris kao primer. Ako je datoteka predviđena za memorijsko mapiranje (koristeći sistemski poziv *mmap*()), Solaris mapira datoteku u adresni prostor procesa. Međutim, ako se datoteka otvara i pristupa joj se pomoću osnovnih sistemskih poziva, kao što su *open*(), *read*() i *write*(), Solaris i dalje memorijski mapira datoteku, međutim, datoteka se mapira u adresni prostor kernela. Bez obzira na to kako se datoteka otvori, Solaris tretira sve U/I operacije na datoteci kao memorijski mapirane, omogućavajući pristup datotekama putem efikasnog memorijskog podsistema.

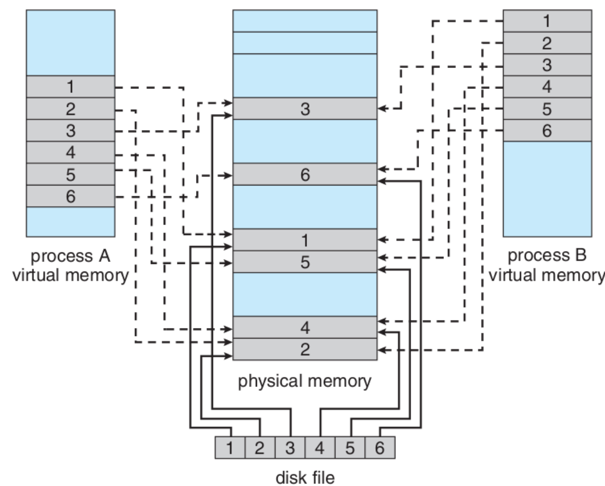
Takođe, moguće je dozvoliti da više procesa istovremeno mapiraju istu datoteku, u cilju deljenja podataka. Upis u datoteku od strane bilo kojeg procesa modifikuje podatke u virtualnoj memoriji i te izmene se mogu vidjeti od strane svih ostalih procesa koji su mapirali isti deo datoteke u svoj adresni prostor. Obzirom na prethodne diskusije o virtualnoj memoriji, trebalo bi biti jasno kako se deljenje memorijski mapiranih delova segmenata memorije implementira: mapa virtualne memorije svakog procesa koji pristupa toj memoriji pokazuje na jedinstvenu stranicu fizičke memorije, odnosno stranicu koja sadrži kopiju bloka sa diska. Ovo deljenje memorije prikazano je na slici 22.

Sistemski pozivi za memorijsko mapiranje mogu takođe podržavati funkciju kopiranja pri upisu, omogućujući procesima da dele datoteku u režimu čitanja, ali da imaju sopstvene kopije podataka koje modifikuju. Da bi pristup deljenim podacima bio sinhronizovan, procesi koji dele podatke mogu koristiti neki od mehanizama za postizanje uzajamne isključivosti opisanih ranije.

Deljena memorija se često, zapravo, implementira korišćenjem memorijski mapiranih datoteka. U ovom scenariju, procesi mogu komunicirati koristeći deljenu memoriju tako što će memorijski mapirati istu datoteku u svoje virtualne adresne prostore.

7.2 Memorijski mapiran U/I

U slučaju U/I periferija, svaki U/I kontroler sadrži registre za smeštanje naredbi i podataka koji se prenose ka uređaju. Obično posebne U/I instrukcije omogućavaju prenos podataka između tih registra i sistemske memorije. Da bi se omogućio pogodniji pristup U/I uređajima, mnoge računarske arhitekture obez-



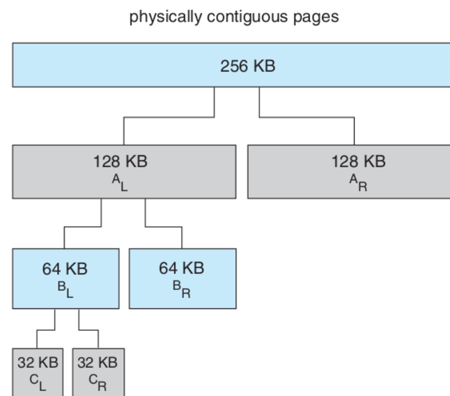
Slika 22: Memorijski mapirane datoteke

beđuju memorijski mapiran U/I. U ovom slučaju se opseg memorijskih adresa rezerviše i preslikavaju se u registre uređaja (kontrolera). Čitanje i upisivanje ovog memorijskog opsega rezultuje prenošenjem podataka do registara uređaja ili iz njih. Ova metoda je pogodna za uređaje koji imaju kratko vreme odziva, kao što su video kontroleri.

Memorijski mapirani U/I pogodni su i za druge uređaje, kao što su serijski i paralelni portovi koji se koriste za povezivanje raznih periferija na računar. CPU prenosi podatke putem ovih uređaja čitanjem i upisom nekolicine registara, koji se nazivaju *U/I port*. Da bi poslao niz bajtova putem memorijski mapiranog serijskog porta, CPU upisuje jedan bajt podataka u registar podataka i postavlja odgovarajući bit u okviru kontrolnog registra da signalizira da je bajt za prenos dostupan. Uređaj preuzima bajt podataka, a zatim briše bit u kontrolnom registru kako bi signalizirao da je spreman za sledeći bajt. Ako CPU koristi *prozivanje* (eng. *polling*) kako bi pratio vrednost bita u kontrolnom registru, izvršavajući se u petlji kako bi proveravao da li je uređaj spreman, ta metoda naziva se *programirani U/I* (eng. *programmed I/O* - PIO). Ako CPU ne vrši prozivanje bita u kontrolnom registru, već umesto toga dobija putem prekida informaciju kada je uređaj spreman za sledeći bajt, kaže se da je prenos podataka kontrolisan prekidom (eng. *interrupt driven*).

8 Alokacija kernel memorije

Kada proces koji se izvršava u korisničkom režimu zahteva dodatnu memoriju, stranice se dodeljuju sa *liste slobodnih okvira* stranica koje održava kernel. Ova lista se obično ažurira od strane algoritma za zamenu stranice, a uglavnom



Slika 23: Alokacija polovljenjem

sadrži slobodne stranice razbacane po fizičkoj memoriji. Takođe, ako korisnički proces zahteva jedan bajt memorije, doći će do interne fragmentacije, jer će proces dobiti ceo okvir memorije na raspolaganje.

Kernel memorija, sa druge strane, se često dodeljuje iz domena slobodne memorije koja se razlikuje od liste koja se koristi da bi se zadovoljili uobičajeni procesi u korisničkom režimu. Postoje dva osnovna razloga za to:

1. Kernel zahteva memoriju za strukture podataka različitih veličina, od kojih su neke manje od stranice. Kao rezultat toga, kernel mora da koristi memoriju na konzervativni način i pokušava da minimizira neiskorišćenu memoriju nastalu usled fragmentacije. Ovo je posebno važno jer većina operativnih sistema ne koristi straničenje u slučaju kernel koda ili kernel podataka;

2. Stranice dodeljene procesima u korisničkom režimu ne moraju nužno da budu u kontinualnoj fizičkoj memoriji. Međutim, određeni hardverski uređaji direktno komuniciraju sa fizičkom memorijom, bez interakcije virtualnog memorijskog interfejsa. Shodno tome, za takvu komunikaciju možda će biti neophodna memorija alocirana na fizički susednim lokacijama.

U nastavku analiziramo dve strategije za upravljanje slobodnom memorijom koja je dodeljena kernel procesima: alokacija polovljenjem i alokacija objekata.

8.1 Alokacija polovljenjem (*Buddy allocation*)

Ovaj metod alokacije dodeljuje memoriju iz segmenata fiksne veličine koji se sastoje od fizički kontinualne memorije. Memorija se dodeljuje alokacijama blokova veličine stepena 2 (4KB, 8KB, 16KB itd). Zahtev za memorijom drugačije veličine zaokružuje se na sledeći dovoljno velik segment. Na primer, zahtev za 11KB dovešće do alokacije segmenta od 16KB.

Razmotrimo jednostavan primer. Pretpostavimo da je veličina memorijskog segmenta u početku 256KB, a kernel zahteva 21KB memorije. Segment je u početku podeljen na dva jednaka segmenta - koje ćemo nazvati A_L i A_R veličine 128 KB. Jedan od ovih segmenata dalje je podeljen na dva pod segmenta od 64 KB - B_L i B_R. Međutim, sledeći segment veličine stepena 2 veći od 21KB iznosi 32KB, tako da je ili B_L ili B_R ponovo podeljen na dva 32KB segmenta, C_L i C_R. Jedan od ovih segmenata veličine 32KB koristi se za ispunjavanje zahteva za memorijom veličine 21 KB. Ova šema je prikazana na slici 23, gde je C_L segment dodeljen kao rezultat zahteva za 21KB. Prednost alokacije polovljenjem je u tome što se brzo mogu kombinovati susedni segmenti kako bi se formirali veći segmenti korišćenjem tehnike poznate kao *spajanje* (eng. *coalescing*). Na slici 23, na primer, kada kernel oslobađa C_L segment koji mu je dodeljen, sistem može spojiti C_L i C_R u segment od 64KB. Ovaj segment, B_L, može, opet, da se spoji sa svojim susednim segmentom B_R i formira segment od 128 KB. Na kraju, možemo završiti sa originalnim segmentom od 256KB, na sličan način.

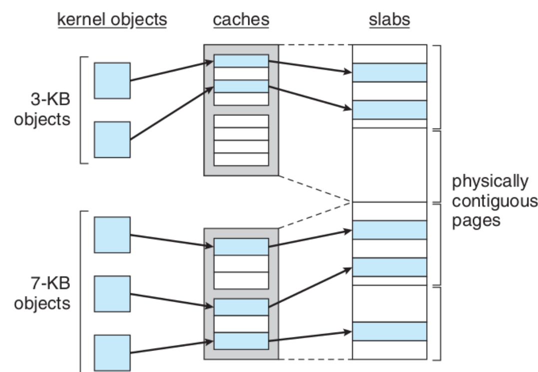
Očigledan nedostatak ove metode alokacije leži u činjenici da će zaokruživanje na segmente veličine sledećeg stepena broja 2 najčešće dovesti do interne fragmentacije unutar dodeljenih segmenata. Na primer, zahtev za memorijom veličine 33 KB može se zadovoljiti samo segmentom od 64 KB. Zapravo, ne možemo nikako garantovati da će manje od 50 procenata dodeljenog segmenta biti izgubljeno usled interne fragmentacije. U sledećem odeljku istražujemo šemu raspodele memorije kod koje se ne gubi memorijski prostor kao posledica fragmentacije.

8.2 Alokacija objekata (*Slab allocation*)

Druga strategija za alociranje kernel memorije je poznata kao alokacija objekata. Objekti se smeštaju u tzv *ploče* (eng. *slab*) sačinjene od jedne ili više fizički povezanih stranica (kontinualna memorija). Keš se sastoji od jedne ili više ploča, pri čemu postoji zaseban keš za svaku jedinstvenu kernel strukturu. Na primer, keš koji se koristi za čuvanje podataka o procesima, keš koji se koristi za datoteke, keš za semafore i tako dalje. Svaki keš je popunjen objektima koji su instance kernel struktura podataka asociranih sa datim kešom. Odnos keša, ploča i objekata prikazan je na slici 24. Slika prikazuje dva kernel objekta veličine 3kB i tri objekta veličine 7kB, keširanih i smeštenih u zasebnim pločama.

Algoritam alokacije objekata koristi keš kako bi smeštao kernel objekte. Kada se keš kreira, određeni broj objekata, inicijalno označenih kao *slobodni*, alocira se za dati keš. Broj objekata u okviru keša zavisi od veličine ploče asocirane sa datim kešom. Na primer ploča veličine 12kB (sačinjena od 3 susedne 4kB stranice), može da skladišti 6 objekata veličine 2kB. Inicijalno, svi objekti u okviru keša označeni su kao *slobodni*, a kada je potreban novi objekat za određenu kernel strukturu, alokator dodeljuje jedan od slobodnih objekata iz keša kako bi ispunio zahtev. Taj objekat se tada označava kao *korišćen*.

Razmatramo scenario u kojem kernel zahteva memoriju kako bi skladištio objekat koji predstavlja desktiptor procesa. U Linux operativnom sistemu, on je predstavljen strukturom *task_struct*, koja zauzima otprilike 1.7kB memo-



Slika 24: Alokacija objekata

rije. Kada Linux kernel kreira novi proces, on zahteva od odgovarajućeg keša memoriju kako bi smestio *task_struct* objekat. Zahtev će biti ispunjen korišćenjem prethodno alociranih objekata strukture *task_struct* koji su označeni kao *slobodni* u odgovarajućoj ploči. U Linux-u, ploča može biti u jednom od tri stanja:

1. Popunjena. Svi objekti u okviru ploče su označeni kao korišćeni;

2. Prazna. Svi objekti u okrivu ploče su označeni kao slobodni;

3. Delimično popunjena. Ploče sadrži kako slobodne tako i korišćene objekte.

Alokator ploče najpre pokušava da odgovori na zahtev korišćenjem *slobodnih* objekata iz delimično popunjene ploče. Ako takav ne postoji, slobodan objekat se dodeljuje iz prazne ploče. Ako, ipak, prazna ploče ne postoji, nova ploče je alocirana iz fizički kontinualne memorije i dodeljena kešu, a memorija za objekat će se alocirati iz te ploče.

Alokacija na ovaj način ima dva benefita:

1. Memorija se ne gubi nepotrebno usled fragmentacije. Fragmentacija nije problem jer svaki jedinstveni kernel objekat ima sebi asocirani keš, a svaki keš se sastoji od jedne ili više ploča podeljenih u blokove memorije veličine objekta za koji su namenjene. Stoga, kada kernel zahteva memoriju za dati objekat, alokator objekata vraća upravo zahtevanu količinu memorije, potrebne da se skladišti dati objekat.

2. Zahtevi za memorijom mogu biti ispunjeni brzo. Shema opisana iznad je posebno efikasna za upravljanje memorijom u sistemu gde se objekti učestalo alociraju i dealociraju, što je upravo slučaj sa kernelom. Ova alokacija i dealokacija može biti skupa u pogledu vremena, ali obzirom da su u ovom slučaju objekti kreirani ranije, oni mogu brzo biti dodeljeni iz odgovarajućeg keša. Dodatno, kada kernel završi sa korišćenjem određenog

objekta i oslobodi ga, on će biti označen kao slobodan i vraćen u keš, nakon čega će momentalno biti dostupan narednim zahtevima od strane kernela.

Linux je originalno koristio *alokaciju polovljenjem* kao mehanizam alokacije kernel memorije, ali počevši od verzije kernela 2.2, Linux koristi *alokaciju objekata* za upravljanje kernel memorijom.